UNIVERSIDADE FEDERAL DE SÃO CARLOS

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

REDUZINDO O CUSTO DO TESTE DE MUTAÇÃO COM BASE NO CONCEITO DE ARCOS PRIMITIVOS

PEDRO HENRIQUE KUROISHI

ORIENTADOR: PROF. DR. AURI MARCELO RIZZO VINCENZI

São Carlos – SP
Fevereiro/2021

UNIVERSIDADE FEDERAL DE SÃO CARLOS

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

REDUZINDO O CUSTO DO TESTE DE MUTAÇÃO COM BASE NO CONCEITO DE ARCOS PRIMITIVOS

PEDRO HENRIQUE KUROISHI

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
Fevereiro/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 Pedro Henrique Kuroishi, realizada em 02/02/2021.

Comissão Julgadora:

Prof. Dr. Auri Marcelo Rizzo Vincenzi (UFSCar)

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Rodrigo Funabashi Jorge (UFMS)

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.

AGRADECIMENTO

Primeiramente, gostaria de agradecer à minha família, principalmente aos meus pais e irmãos por todo o suporte durante toda essa trajetória. Em todos os momentos difíceis, vocês me incentivaram e me fizeram seguir em frente.

Um agradecimento especial ao meu orientador, Prof. Dr. Auri Marcelo Rizzo Vincenzi pela orientação, amizade e ensinamento durante todo trabalho.

Ao Prof. Dr. José Carlos Maldonado do ICMC/USP, por toda ajuda durante o trabalho.

Ao professor Prof. Dr. Rodrigo Funabashi Jorge da FACOM/UFMS, pelo ensinamento e incentivo durante a graduação.

A todos do LaPES/UFSCar, pela amizade do dia-a-dia.

A todos os amigos de Campo Grande-MS e, principalmente, aos amigos do grupo ”P. Avança”. *”Me desculpa, não me entenda mal”.*

A todos os docentes, técnicos e funcionários do PPGCC/UFSCar.

À CAPES pelo apoio financeiro.

It was. It will never be again. Remember.

Paul Auster

RESUMO

Contexto: O teste de software é uma atividade fundamental que auxilia na garantia de qualidade de software. Diversas técnicas e critérios auxiliam o testador a derivar e avaliar requisitos de teste. O teste de mutação é um critério do teste baseado em defeitos bastante utilizado para avaliar a qualidade de conjuntos de teste. Entretanto, o alto custo computacional para gerar e executar os mutantes, além da existência de mutantes equivalentes, dificulta uma ampla utilização do critério em situações práticas. Neste contexto, torna-se necessário propor diferentes abordagens com o objetivo de viabilizar o teste de mutação. ***Objetivos***: O objetivo deste trabalho é propor e avaliar uma abordagem de redução do custo do teste de mutação que engloba teste de mutação e o conceito de arcos primitivos, utilizado no contexto dos critérios de fluxo de controle. De forma geral, o objetivo é verificar se ao executar apenas os mutantes localizados no conjunto de arcos primitivos de um programa em teste, é possível reduzir o número de mutantes executados e manter um alto escore de mutação. Outro ponto do trabalho, está relacionado aos mutantes minimais, ou seja, o objetivo é verificar se é possível relacionar a geração dos mutantes minimais no contexto dos arcos primitivos. Caso as hipóteses se confirmem, é possível derivar novos critérios de teste a partir dos resultados obtidos. ***Método***: Para condução deste trabalho, foi realizado um estudo experimental para avaliar a validade da abordagem proposta. Para isso, utilizou-se um conjunto de 29 programas em C de diversos domínio e empregados em diversos estudos envolvendo mutação. ***Resultados***: Os resultados obtidos mostram que foi possível reduzir o número de mutantes e preservar um alto escore de mutação. Além disso, observou-se uma alta concentração de mutantes minimais nos arcos primitivos. Ao comparar a abordagem proposta com uma abordagem já existente, os resultados mostram que a mutação nos arcos primitivos foi superior à mutação aleatória. Por fim, a partir dos resultados, foi possível definir novos critérios de teste envolvendo teste de mutação e arcos primitivos. ***Conclusão***: Embora seja um critério bastante poderoso para detectar defeitos, é notório que o teste de mutação necessita de melhorias para uma maior utilização na indústria. A abordagem proposta pode guiar novos trabalhos, uma vez que os resultados mostram que combinar teste de mutação e arcos primitivos apresentou-se como uma alternativa promissora. Neste sentido, ampliar o experimento com diferentes conjuntos de teste e avaliá-los em programas mais complexos, de nível industrial, pode trazer novos resultados e, assim, apresentar novas contribuições para a comunidade.

Palavras-chave: Teste de Mutação, Análise de Fluxo de Controle, Arcos Primitivos, Redução de Custo, Mutantes Equivalentes, Mutantes Minimais, Critérios de Teste

ABSTRACT

Context: Software testing plays an important role in quality assurance. Testing techniques and criteria help the tester to develop and assess test suites. Mutation testing is a fault-based testing criterion commonly used to evaluate the quality of test suites. However, a high computational cost to generate and execute the mutants and the existence of equivalent prevent mutation testing to be widely applied in practical situations. In this sense, it is important to propose studies to overcome such problems. ***Objective***: This work presents an approach that combined mutation testing and the concept of primitive arcs, used in the context of control-flow testing criteria, to reduce the cost of mutation testing. Overall, the goal was to verify if the execution of a subset of mutants located on the primitive arcs of a program under testing reduces the number of mutants and maintains a high mutation score. Next, this work evaluated the relationship between minimal mutants and primitive arcs. If both hypotheses confirm, it is possible to design new testing criteria based on the results obtained. ***Method***:This work presented an experimental study to evaluate the proposed approach. To carry out the experiment, this work considered a set of 29 programs in C already used in other studies involving mutation testing. ***Results***: The results showed that the approach reduced the number of mutants and achieved a high mutation score. Besides, the results showed that the primitive arcs concentrated a high percentage of minimal mutants. Then, the approach was compared to random mutation. The results showed that mutation on primitive arcs was slightly better than random mutation. Finally, this work presents a set of testing criteria based on the results obtained. ***Conclusion***: Although powerful to detect faults, mutation testing still requires some improvements to large use in industry. The proposed approach may guide further works once the results obtained showed that the combination of mutation testing and primitive arcs might be a promising alternative. Thus, a broad experimental study using industry level programs and different test suites, may bring different results and hence, bring some contribution to the community.

Keywords: Mutation Testing, Control Flow-Analysis, Primitive Arcs, Cost Reduction, Equivalent Mutants, Minimal Mutants, Testing Criterion

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contextualização e Motivação

Atualmente, a tecnologia possui uma grande influência na sociedade. É possível notar uma crescente dependência das pessoas por produtos tecnológicos, uma vez que diversos desses produtos são empregados na realização de tarefas triviais do dia a dia. Neste sentido, torna-se necessário aprimorar o processo de desenvolvimento de software com objetivo de entregar produtos de alta qualidade.

Em que pese o considerável avanço na área de Engenharia de Software, o produto em desenvolvimento está suscetível a diversos defeitos que podem ser ocasionados a partir de uma má interpretação dos requisitos ou advindo de falhas humanas durante a etapa de codificação. Por esse motivo, o teste de software é fundamental para garantir a qualidade do software, uma vez que a principal finalidade é detectar a presença de defeitos, minimizando as falhas que podem ocorrer durante o processo de produção (Myers et al., 2011).

Porém, testar não é uma atividade trivial. Harrold (2000) estimou que o tempo despendido para testar um software corresponde a, aproximadamente, 50% do tempo total gasto para desenvolver o produto. Neste contexto, é importante associar esta atividade em todo o ciclo de desenvolvimento, desde a fase de definição dos requisitos até a etapa de *release* do produto, visto que quanto mais cedo um defeito for detectado, menor será o custo de sua correção (Boehm e Basili, 2001).

Ao longo dos anos, diversas técnicas e critérios vêm sendo desenvolvidos para que os testes sejam realizados de forma sistemática, auxiliando na geração e avaliação de casos de teste que sejam capazes de detectar o maior número de defeitos possíveis em tempo hábil. Dentre as diversas técnicas e critérios propostos destaca-se o Teste de Mutação (DeMillo et al., 1978; Hamlet, 1977), que é um critério de teste baseado em defeitos bastante utilizado para avaliar a qualidade de um dado conjunto de teste.

De forma geral, o objetivo do teste de mutação é desenvolver casos de teste para matar *mutantes* – versões alternativas do programa original com um simples defeito injetado artificialmente. Se um caso de teste for capaz de detectar este defeito, o mutante é considerado *morto*. Em alguns casos, nenhum caso de teste é capaz de apontar as diferenças entre comportamento do mutante e do programa original e, desse modo, o mutante pode ser classificado como equivalente – apresenta a mesma funcionalidade, embora possua diferenças sintáticas em relação ao programa original. Observa-se que esta tarefa de classificação de mutantes equivalentes pode onerar o teste de mutação, visto que, em geral, é uma atividade realizada de forma manual. Ao final do processo é computado o escore de mutação dado pelo número total de mutantes mortos sobre o número total de mutantes não-equivalentes e, em geral, é utilizado para avaliar a adequação de um dado conjunto de teste.

Neste sentido, um elevado custo computacional para geração e execução dos mutantes, além da necessidade de classificar os mutantes equivalentes são duas grandes desvantagens do teste de mutação. Deste modo, nota-se que este critério necessita de melhorias para uma maior utilização em situações práticas. Assim, diversos pesquisadores despendem esforços apresentando novas abordagens com o objetivo de tornar o teste de mutação viável.

Os objetivos gerais do trabalho são apresentados a seguir.

1.2 Objetivos Gerais

Ao longo dos anos, diversos estudos relacionados ao teste de mutação vêm sendo realizados buscando torná-lo viável em diversas situações práticas. Neste sentido, o presente trabalho propõe e avalia uma forma alternativa de redução do custo do teste de mutação.

O objetivo deste trabalho é apresentar uma abordagem que combina teste de mutação e informação sobre o fluxo de controle do programa. Neste caso, o estudo verifica se ao executar apenas mutantes localizados em partes específicas do código, dado pelo conjunto de arcos primitivos (Chusho, 1987) de um programa em teste, é possível reduzir o número de mutantes que serão executados e ainda preservar um alto escore de mutação. Além disso, o trabalho verifica se a abordagem também é capaz de reduzir o número de mutantes equivalentes.

Doravante, o estudo verifica se é possível relacionar o conceito de mutantes minimais e arcos primitivos, conforme sugerido por Delamaro et al. (2018). Neste caso, o trabalho avalia se os arcos primitivos tendem a concentrar um alto número de mutantes minimais.

Para avaliar a abordagem, foi realizado um estudo experimental com programas em C, verificando a redução do custo em termos do número de mutantes e a eficácia dada pelo escore de mutação. Para validar as hipóteses apresentadas, foram executados testes estatísticos e, em seguida, a abordagem foi comparada com uma já existente: a mutação aleatória.

Por fim, a partir dos resultados obtidos, o trabalho apresenta uma definição preliminar de novos critérios de teste que envolvem teste de mutação e arcos primitivos.

1.3 Organização da Dissertação

O presente capítulo apresentou uma breve contextualização e motivação, além dos objetivos gerais para a condução deste estudo. Os próximos capítulos estão organizados da seguinte forma:

- O Capítulo 2 apresenta a fundamentação teórica, abordando uma visão geral sobre os principais conceitos empregados no trabalho. Inicialmente, são apresentados conceitos básicos de teste de software, bem como as técnicas e critérios. A seguir, o teste de mutação é detalhado, mostrando as principais desvantagens deste critério. Por fim, o capítulo apresenta os conceitos de mutantes minimais e arcos primitivos.
- O Capítulo 3 apresenta a revisão bibliográfica, mostrando os trabalhos relacionados a esta dissertação.
- O Capítulo 4 detalha o estudo experimental realizado. Inicialmente, são apresentadas as questões de pesquisa do trabalho. Em seguida, a estrutura experimental é definida, mostrando os programas utilizados e os passos necessários para execução do experimento.
- O Capítulo 5 apresenta e discute os resultados obtidos do estudo experimental e as respostas para as questões de pesquisa.
- O Capítulo 6 compara a abordagem proposta no trabalho e a mutação aleatória.
- O Capítulo 7 apresenta uma definição preliminar de novos critérios de teste. Além disso, o capítulo apresenta a relação de inclusão entre os critérios e uma possível estratégia de aplicação dos mesmos.
- O Capítulo 8 conclui a dissertação, apresentando as limitações do estudo e possíveis trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

O presente capítulo apresenta a fundamentação teórica, abordando os principais conceitos empregados no trabalho. Inicialmente, são apresentados conceitos básicos de teste de software, bem como as técnicas e critérios. Em seguida, a Seção 2.3 detalha o teste de mutação, que é o foco deste trabalho. Já a Seção 2.3.2 e a Seção 2.4 apresentam conceitos dos mutantes minimais e arcos primitivos, respectivamente, e que foram abordados neste trabalho.

2.2 Teste de Software: Conceitos e Terminologias

O teste de software é uma atividade fundamental que auxilia na garantia da qualidade do produto que está sendo desenvolvido (Pressman, 2010). Ao contrário do senso comum, o teste de software tem como objetivo detectar eventuais defeitos que possam ocorrer durante o ciclo de desenvolvimento (Myers et al., 2011). Deste modo, quanto menor a quantidade de defeitos detectados, maior será a confiança no produto desenvolvido. Inicialmente, para uma melhor compreensão de conceitos básicos que serão apresentados, é importante compreender algumas terminologias comumente utilizadas. O padrão *IEEE 24765-2017* (IEEE, 2017) define estas terminologias da seguinte forma:

- **Defeito (*Fault*)**: passo, processo ou definição de dado incorreto.
- **Erro (*Error*):** uma ação que produz um resultado incorreto do software.
- **Falha (*Failure*):** produção de uma saída incorreta com relação à especificação.

Em geral, a atividade de teste engloba três fases (Delamaro et al., 2016): *teste de unidade*, *teste de integração* e *teste de sistema*. No teste de unidade, o objetivo é

verificar a menor parte do software, que podem ser funções, métodos, procedimentos ou classes, identificando erros de lógica ou de implementação do algoritmo. Em seguida, é realizado o teste de integração. Nesta fase, as unidades são integradas visando identificar os defeitos advindo desta integração. Por fim, o teste de sistema garante que o software funcionará como um todo, ou seja, todas as funcionalidades estarão de acordo com a especificação. Alguns autores (Copeland, 2003) definem uma quarta fase, o teste de aceitação, realizado ao final do ciclo de teste e que objetiva verificar se o software desenvolvido está de acordo com as necessidades do usuário.

O processo para testar um software, conforme ilustrado na Figura 2.1, engloba quatro etapas (Delamaro et al., 2016; Maldonado, 1991): *planejamento*, *projeto de casos de teste*, *execução dos casos de teste* e *análise dos resultados dos testes*. Na fase de planejamento é definido um plano de teste que contempla informações sobre como os testes serão realizados, os recursos, técnicas e critérios que serão aplicados, além das métricas para avaliar a qualidade do software (Maldonado, 1991). A próxima etapa consiste na criação dos casos de teste que serão executados no programa. Um caso de teste é um par ordenado $(d, S(d))$, onde $d \in D$ corresponde a um dado de teste do domínio de entrada D e $S(d)$ é a saída esperada de d. Ademais, os casos de teste são executados no programa em teste e um *oráculo de teste* analisa se a saída $S(d)$ está de acordo com a saída esperada (Howden, 1978).

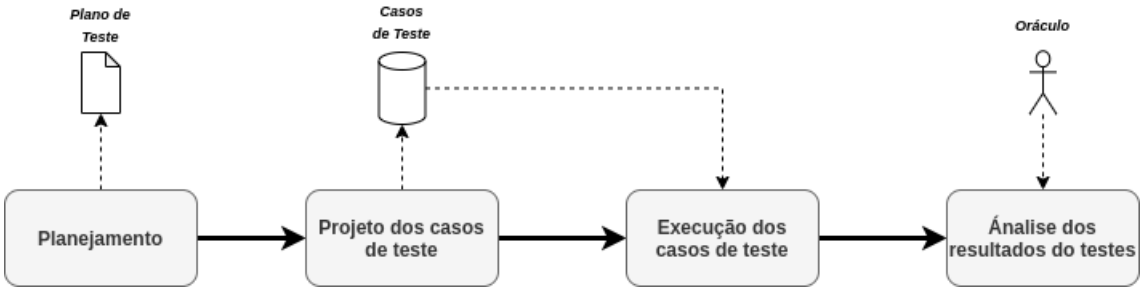


Figura 2.1: Processo de teste simplificado adaptado de Delamaro et al. (2016).

Em um cenário ideal, o programa seria testado com todas as entradas possíveis do domínio D. Entretanto, como o número de entradas pode ser infinito, testar o software à exaustão é uma prática, em geral, inviável. Desta forma, as técnicas e critérios de teste fornecem subsídios para que o teste seja realizado de forma sistemática, a fim de prover software de alta qualidade.

2.2.1 Teste Funcional

A técnica de teste funcional, também conhecida como teste caixa preta, tem como objetivo encontrar circunstâncias nas quais o programa em teste não está de acordo com sua especificação (Myers et al., 2011). O software é visto como uma caixa preta e,

neste caso, não existe a necessidade de compreender a implementação e a estrutura interna do software.

Copeland (2003) define um processo para aplicação do teste funcional, conforme ilustrado de forma resumida na Figura 2.2. O primeiro passo é analisar a especificação do programa. Em seguida, desenvolvem-se casos de teste válidos e as saídas esperadas para estas entradas. Por fim, o conjunto de teste é executado e a saída obtida é comparada e analisada com a saída esperada.

Figura 2.2: Exemplo do teste caixa preta.

Ademais, uma desvantagem da técnica funcional é que, em geral, não é possível afirmar com certeza se o software já foi totalmente testado, uma vez algumas estruturas podem não ser cobertas pelos casos de teste (Copeland, 2003).

Em relação aos critérios da técnica funcional, destacam-se: *particionamento da classe de equivalência* e *análise do valor limite*. O primeiro critério tem como finalidade dividir o conjunto de entrada em número finito de classes de equivalência, no qual um caso de teste pertencente a uma dessas partições, equivale a qualquer componente desta classe (Myers et al., 2011). Ou seja, se um caso de teste for capaz de detectar um defeito, então todos os elementos desta classe detectarão o mesmo defeito (Copeland, 2003).

Já o critério da análise do valor limite é considerado o complemento ao particionamento de classe de equivalência, entretanto, ao invés de escolher um valor aleatório, escolhem-se elementos nos limites (inferior e superior) da classe, uma vez que, em geral, os defeitos se concentram nos limites das partições (Copeland, 2003).

Reid (1997) realizou um estudo analisando empiricamente a efetividade de três critérios de teste do teste funcional, dentre eles particionamento de equivalência e análise do valor limite. Em um dos resultados obtidos verificou-se que a análise do valor limite possui uma maior efetividade em relação ao primeiro, entretanto necessita do triplo de casos de teste. Um outro resultado interessante é que, uma maior quantidade de casos de teste não influencia no custo de aplicação de ambos os critérios, uma vez que o custo está relacionado à identificação das classes de equivalência.

2.2.2 Teste Estrutural

No teste estrutural, ou teste caixa branca, é necessário compreender a lógica interna e a implementação do programa para derivar os casos de teste (Myers et al., 2011; Pressman, 2010). Assim como no teste funcional, Copeland (2003) define um processo geral para realização do teste estrutural: i) análise da implementação do software; ii) identificação dos caminhos; iii) definição dos casos de teste que irão executar os caminhos identificados e das saídas esperadas; iv) execução dos casos de teste; v) comparação das saídas obtidas com a saída esperada. A Figura 2.3 ilustra, resumidamente, este processo.

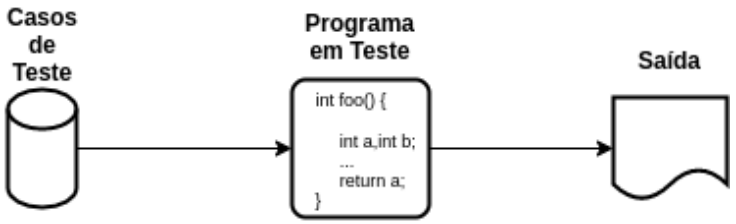


Figura 2.3: Exemplo do teste caixa branca.

Em relação aos critérios do teste estrutural, destacam-se: *critérios de teste baseados em fluxo de controle* e *critérios de teste baseados em fluxo de dados*.

No critério de teste baseado em fluxo de controle, o programa em teste é transformado em um grafo fluxo de controle (*GFC*), no qual cada nó corresponde a uma ou mais instruções que serão executadas em sequência e cada aresta representa o fluxo de controle, ou seja, um possível desvio de um nó para outro ou transferência de controle entre os nós. Em geral, os critérios baseados em fluxo de controle compreendem às seguintes condições (Delamaro et al., 2016):

1. *Todos-Nós*: todos os nós do grafo fluxo de controle devem ser exercitados ao menos uma vez.

2. *Todas-Arestas*: todas as arestas do grafo fluxo de controle devem ser exercitadas ao menos uma vez.

3. *Todos-Caminhos*: todos os caminhos do grafo fluxo de controle devem ser exercitados ao menos uma vez.

Como o número de caminhos pode ser infinito e percorrê-los em sua totalidade não garante que todos os erros sejam detectados, é necessário um critério capaz de selecionar um subconjunto destes caminhos que sejam efetivos (Rapps e Weyuker, 1985).

A segunda família de critérios, baseada em fluxo de dados e apresentada por Rapps e Weyuker (1982), analisa os valores associados às variáveis e como esta

associação irá afetar a execução do programa. Assim, as informações de fluxo de dados são utilizadas para derivar os casos de teste (M. Herman, 1976). Cada uma das ocorrências de variáveis é classificada de acordo com sua definição (*def*), uso computacional (*c-use*) e uso de predicado (*p-use*). Em seguida, define-se o grafo *def-use*, no qual cada nó associa-se aos conjuntos *def* e *c-use* e cada uma das arestas corresponde ao conjunto *p-use*. Ademais, foram definidos uma família de critérios para seleção dos caminhos sendo os principais (Delamaro et al., 2016): *todas-definições*, *todos-usos* (e suas variações) e *todos-du-caminhos*. Maldonado (1991) definiu uma família de critérios Potenciais-Usos, estendendo a família de critérios apresentada por Rapps e Weyuker (1982).

Por fim, a técnica estrutural pode ser utilizada com uma técnica complementar à técnica funcional, visto que ambos os critérios cobrem tipos distintos de defeitos e, além disso, podem ser aplicados em fases distintas do teste (Pressman, 2010).

2.2.3 Teste Baseado em Defeitos

Segundo DeMillo et al. (1987), a técnica de teste baseado em defeitos tem como principal objetivo o desenvolvimento de casos de teste capazes de revelar a presença e a ausência de alguns defeitos específicos. Dentre os critérios do teste baseado em defeitos, destaca-se o teste de mutação que será abordado com mais detalhes na Seção 2.3.

2.3 Teste de Mutação

O teste de mutação, inicialmente proposto por Hamlet (1977) e DeMillo et al. (1978), é um critério do teste baseado em defeitos comumente utilizado para avaliar a qualidade de um conjunto de teste. Este critério baseia-se em duas hipóteses: **Hipótese do Programador Competente** e **Efeito do Acoplamento**. A primeira hipótese garante que o programador escreve programas corretos ou próximos de estarem corretos (Acree et al., 1979). Já no Efeito do Acoplamento, se o caso de teste é capaz de detectar defeitos simples, conclui-se que ele também é capaz de detectar defeitos mais complexos (DeMillo et al., 1978). Desta forma, supondo a validade de ambas hipóteses, tem-se que o Teste de Mutação é um critério bastante interessante, uma vez que defeitos mais complexos podem ser eliminados ao corrigir os defeitos mais simples (Acree, 1980).

A Figura 2.4 ilustra de forma simplificada o processo de aplicação do teste de mutação. Inicialmente, o conjunto de teste T é executado no programa P que será testado. Se, após a execução de T, o resultado obtido diferir do resultado esperado,

conclui-se que o programa P possui um defeito e precisa ser corrigido. O próximo passo consiste na geração do conjunto de **mutantes** M de P. Um mutante $m_i \in M$ é uma versão modificada de *P* com uma simples alteração sintática injetada artificialmente. Ressalta-se que tais alterações são causadas por um conjunto de operadores de mutação que, em geral, representam erros comuns que os programadores cometem durante a fase de desenvolvimento. A Figura 2.5 mostra um exemplo de mutante que substitui o operador aritmético de soma (*+*) pelo operador aritmético de multiplicação (***).

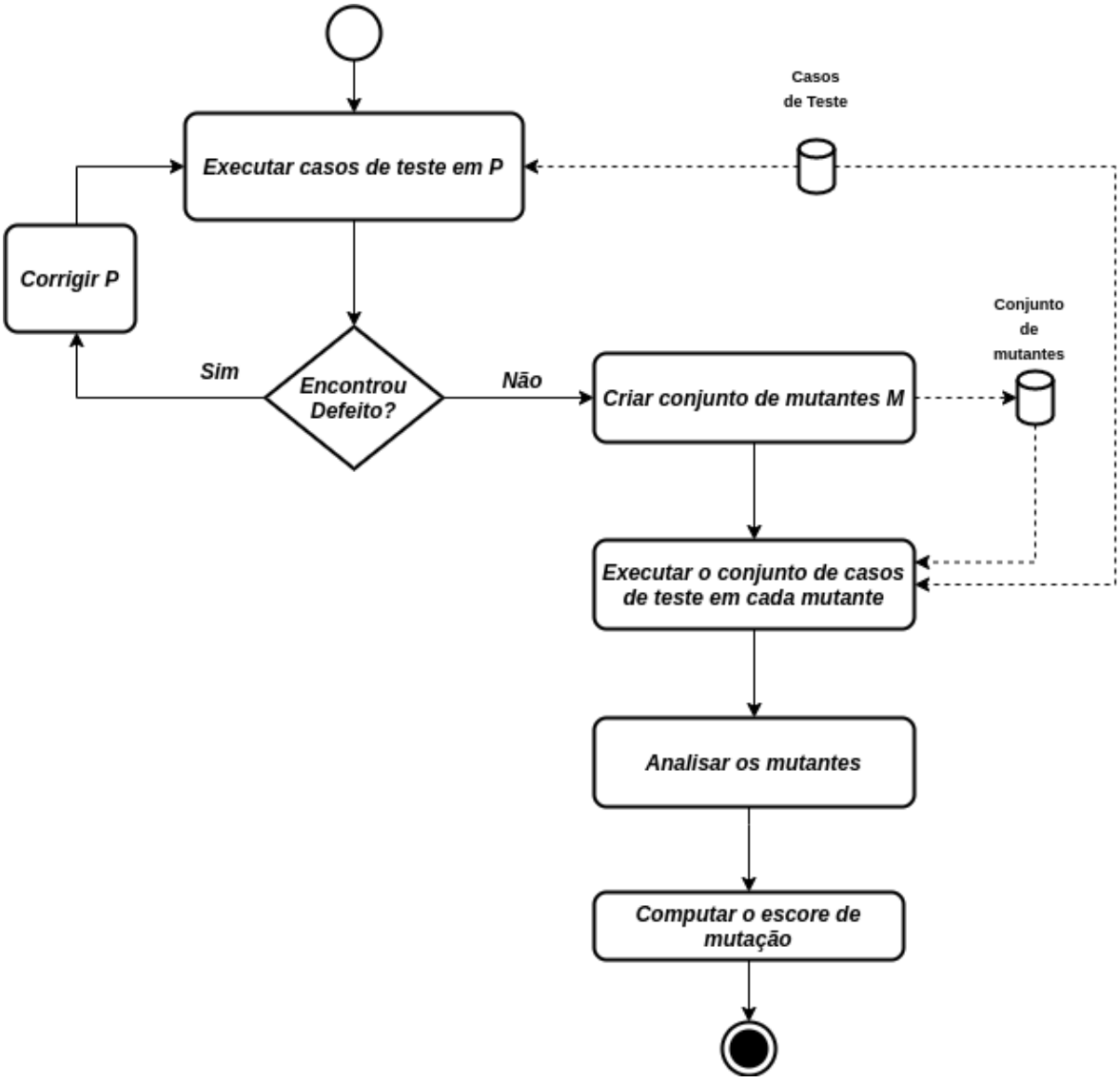


Figura 2.4: Processo simplificado do teste de mutação.

Após a criação do conjunto de mutantes M, o conjunto de casos de teste T é executado em cada mutante $m_i \in M$, para $1 \leq i \leq |M|$, e os resultados de cada mutante são analisados. Ao executar $t_j \in T$, para $1 \leq j \leq |T|$, em cada m_i, verificam-se os seguintes casos:

Código Original	Exemplo de Mutante

`int sum(int x[], int size)`
`{`
`  int s = 0;`
`  int i;`
 	
`  for(i = 0; i < size; i++) {`
`  s = s + x[i];`
`  }`
 	
`  return s;`
`}`
	`int sum(int x[], int size)`
	`{`
	`  int s = 0;`
	`  int i;`
	
	`  for(i = 0; i < size; i++) {`
	`  s = s * x[i];`
	`  }`
	
	`  return s;`
	`}`

Figura 2.5: Exemplo de um mutante.

- **Caso 1:** Se $m_i(t_j) \neq P(t_j)$, então o caso de teste t_j foi capaz de detectar a mutação e, portanto, o mutante t_i é dito *morto*.
- **Caso 2:** Se $m_i(t_j) = P(t_j)$, então o caso de teste t_j não foi capaz de detectar a mutação e, portanto, o mutante t_i é dito *vivo*.

Quando um mutante m_i é classificado como vivo, deve-se avaliar se é necessário criar casos de teste adicionais com intuito de detectar tal mutante ou verificar se o mutante é *equivalente* – sintaticamente diferente, porém semanticamente iguais. Visto que nenhum caso de teste é capaz de distinguir o comportamento entre o programa original e os mutantes equivalentes, estes podem ser descartados da análise. A Figura 2.6 demonstra um exemplo de mutante equivalente, no qual um operador de mutação substitui o operador de atribuição (=) por um operador aritmético com atribuição (*=).

Código Original	Exemplo de Mutante Equivalente

`int sum(int x[], int size)`
`{`
`  int s = 0;`
`  int i;`
 	
`  for(i = 0; i < size; i++) {`
`  s = s + x[i];`
`  }`
 	
`  return s;`
`}`
	`int sum(int x[], int size)`
	`{`
	`  int s = 0;`
	`  int i;`
	
	`  for(i *= 0; i < size; i++) {`
	`  s = s + x[i];`
	`  }`
	
	`  return s;`
	`}`

Figura 2.6: Exemplo de um mutante equivalente.

Ao final do processo, o valor do escore de mutação é computador a partir da Equação 2.1:

$$MS = \frac{\#\ Mutantes\ Mortos}{\#\ Total\ de\ Mutantes - \#\ Mutantes\ Equivalentes} \tag{2.1}$$

O escore de mutação é um valor no intervalo [0,1] e é utilizado para medir a adequação do conjunto de teste. Assim, quanto maior for o escore obtido, maior será a confiança em relação conjunto de teste. Analogamente, quanto menor o escore menor será a confiança em relação ao conjunto de teste.

Um ponto importante a ser ressaltado é que, em geral, só é possível obter o escore de mutação igual à 1 após a análise dos mutantes vivos, uma vez que é necessário a

identificar os mutantes equivalentes e descartá-los da análise (Vincenzi et al., 2006). Quando o escore atinge o valor de 1, diz-se que o conjunto de teste é adequado ao teste de mutação.

2.3.1 Custo do Teste de Mutação

O teste de mutação é considerado um critério bastante poderoso para detectar defeitos. Embora, algumas iniciativas recentes abordem a utilização do critério em situações práticas (Delgado-Pérez et al., 2018; Petrovic e Ivankovic, 2017; Petrovic et al., 2018), o teste de mutação ainda necessita de algumas otimizações para uma maior aplicação na indústria. Sabe-se que o custo computacional para gerar e executar os mutantes, além da existência de mutantes equivalentes são duas grandes desvantagens do teste de mutação.

Como descrito anteriormente, um mutante é dito equivalente quando apresenta a mesma funcionalidade do programa original em teste. O estudo realizado por Budd e Angluin (1982) mostrou que o problema da equivalência entre programas é *indecidível* e, desta forma, é necessário que a classificação de mutantes equivalentes seja realizada por meio de heurísticas ou de forma manual, tornando a atividade relativamente custosa. No estudo de Schuler e Zeller (2010), os autores avaliaram que 45% dos mutantes que permanecem vivos são equivalentes e, além disso, o tempo para classificar se um único mutante é equivalente é de aproximadamente 15 minutos.

Em relação ao custo computacional do teste de mutação, nota-se que o número de mutantes gerados pode ser bastante elevado até para programas simples e de baixa complexidade. Neste caso, um alto número de mutantes pode elevar consideravelmente o custo computacional, uma vez que o conjunto de teste é executado em cada um dos mutantes gerados (Offutt et al., 1993).

Ao longo dos anos, diversas pesquisas têm sido realizadas visando otimizar o teste de mutação, apresentando diferentes métodos para reduzir os custos atrelados a este critério. Offutt e Untch (2001) classificaram os estudos relacionados à redução do custo do teste de mutação em três categorias: *do fewer*, *do faster* e *do smarter*. As técnicas *do fewer* visam reduzir o número de mutantes que serão executados. Já as técnicas *do faster* focam em gerar e executar os mutantes o mais rápido possível. E as técnicas *do smarter* focam em reduzir o custo computacional do teste de mutação.

Algumas abordagens *do fewer* clássicas buscam reduzir o número de mutantes gerados a partir de um subconjunto reduzidos de operadores de mutação. Por exemplo, na *mutação seletiva* (Mathur, 1991; Offutt et al., 1993), escolhe-se um subconjunto de N operadores de mutação que geram um conjunto M de mutantes com a mesma representatividade do conjunto de mutantes gerados a partir de todos os operadores de mutação. Seguindo nesta linha, a abordagem dos *operadores essenciais* (Barbosa

et al., 2001; Offutt et al., 1996), assemelha-se à mutação seletiva, porém o método para gerar o conjunto de operadores suficientes possui uma maior complexidade. Por fim, na *mutação aleatória* (Wong e Mathur, 1995), o testador seleciona um subconjunto de mutantes que serão executados, baseado numa porcentagem predefinida.

A abordagem presente neste trabalho pode ser classificado como *do fewer* uma vez que objetiva reduzir o número de mutantes que serão executados. Entretanto, ao invés de apresentar uma abordagem baseada em um subconjunto específico de operadores de mutação, o foco está apenas os mutantes localizados em partes específicas no código serão executados, dado pelo conjunto de arcos primitivos (Chusho, 1987) do programa em teste. O conceito de arcos primitivos será abordado com mais detalhes na Seção 2.4.

2.3.2 Mutantes Minimais

Ammann et al. (2014) propuseram uma forma de computar um conjunto de mutantes minimais. A ideia é fornecer um limite teórico no número de mutantes necessários para manter a adequação de um conjunto de teste. Neste contexto, a abordagem visa remover os mutantes considerados *redundantes*, ou seja, aqueles mutantes que ao serem removidos não afetam o escore de mutação.

Os autores apresentaram o conceito de relação de inclusão dinâmica que consiste basicamente em: dado um conjunto de teste T, um conjunto de mutantes M e $m_i, m_j \in M$, dois mutantes quaisquer. Então, o mutante m_i inclui dinamicamente o mutante m_j, se todos os casos de teste que matam m_i também matam m_j. Assim, um mutante é dito redundante se é incluído dinamicamente por outro mutante. Por sim, um conjunto de mutantes minimais é o conjunto que não possui nenhum mutante redundante.

Além disso, Ammann et al. (2014) propuseram um algoritmo para calcular o conjunto de mutantes minimais. Seja M um conjunto de mutantes e T um conjunto de teste adequado. Define-se uma matriz binária $|M| \times |T|$, denominada *score function*[1], que especifica quais casos de teste mata um mutante. Doravante, analisa-se a *score function* verificando os mutantes que não são incluídos dinamicamente e, em seguida, são adicionados ao conjunto de mutantes minimais. No caso em que um mutante possui a mesma *score function* de outro mutante, ambos são denominados indistinguíveis e, neste caso, apenas um será adicionado ao conjunto de mutantes minimais, indicando que o conjunto de mutantes minimais pode não ser único.

[1]O conceito de *score function* foi denominado por Ammann et al. (2014), porém, a *score function* também pode ser vista como uma *killing matrix*, uma vez que ambas apresentam as informações de quais casos de teste matam cada um dos mutantes

Vale ressaltar que para definir o conjunto de mutantes minimais é necessário executar, ao menos uma vez, todos os casos de teste de T em todos os mutantes de M e, neste caso, pode acarretar em um aumento do custo do teste de mutação.

O presente trabalho aborda os mutantes minimais conforme proposto por Delamaro et al. (2018), verificando se existe uma relação entre a geração dos mutantes minimais e os arcos primitivos. Ou seja, de forma geral, o objetivo é verificar se existe uma alta concentração de mutantes minimais nos arcos primitivos.

2.4 Arcos Primitivos

O conceito de arcos primitivos propostos por Chusho (1987) é definido como os arcos que nunca são herdeiros de outros arcos do grafo fluxo de controle. Um arco é considerado herdeiro sempre que sua execução ocorrer após a execução de outro arco. Considere a e b arcos de um grafo fluxo de controle. Se todo caminho que passar por a sempre incluir o arco b, então b é denominado arco herdeiro e a arco ancestral. Note que b irá herdar as informações sobre execução de a.

O algoritmo de Chusho (1987) transforma um grafo fluxo de controle em um grafo reduzido de herdeiro aplicando um conjunto de regras para eliminar todos os arcos herdeiros até obter um grafo contendo apenas arcos primitivos – ou arcos essenciais.

Antes de apresentar as quatro regras para derivar um grafo reduzido de herdeiros, é necessário destacar alguns conceitos fundamentais para desenvolver o algoritmo:

- O programa em teste é transformado em um grafo fluxo de controle denotado por $G(N, A)$, sendo N o conjunto de nós e A o conjunto de arcos. Um arco é representado por um par ordenado do tipo (x, y), com $x, y \in N$.

- Sendo x um nó, então $IN(x)$ corresponde ao número de arcos que entram no nó x. Analogamente, $OUT(x)$ equivale ao número de arcos que saem de x. Se $IN(x) = 0$, então o nó é um nó de entrada e, se $OUT(x) = 0$, então o nó é de saída.

- Seja $DOM(x)$ e $IDOM(x)$ os conjuntos de dominadores e dominadores inversos de x, respectivamente. Um nó y é dominador de x, se todos os caminhos de um nó de entrada até x inclui y. Já um nó y é dominador inverso de x, se todos os caminhos de x até um nó de saída inclui y.

- As regras devem ser aplicadas em um grafo dirigido $G(N, A)$ tal que $x, y \in N \wedge x \neq y \wedge (x, y) \in A$.

Munido destes conceitos é possível definir as quatro regras para aplicação do algoritmo de Chusho:

1. Se $IN(x) \neq 0$ e $OUT(x) = 1$, então elimina-se o arco (x, y) de A e x e y são unidos.

2. Se $IN(y) = 1$ e $OUT(y) \neq 0$, então elimina-se o arco (x, y) de A e os nós x e y são unidos.

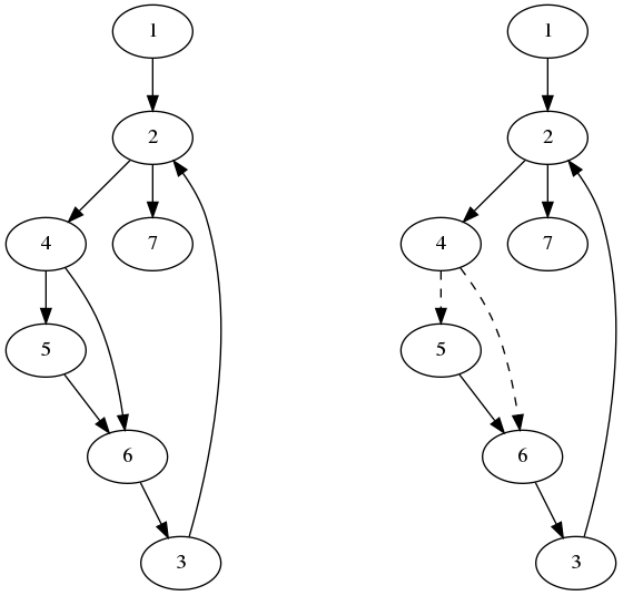
3. Se $OUT(x) \geq 2$ e $x \in IDOM(w), \forall w \in \{w|(x, w) \in A \wedge w \neq y\}$, então elimina-se o arco (x, y) de A e x e y são unidos.

4. Se $IN(y) \geq 2$ e $y \in DOM(w), \forall w \in \{w|(w, y) \in A \wedge w \neq x\}$, então elimina-se o arco (x, y) de A e x e y são unidos.

Na implementação do algoritmo, as regras são aplicadas de forma sequencial. Nos primeiros passos executam-se as regras 1 e 2, respectivamente, enquanto houver arcos que as satisfaçam. Em seguida, assinalar cada arco (x, y) como uma marca de herdeiro que satisfaça a regra 3, se houver pelo menos um arco sem esta marca que entram em x ou entre os arcos que compõem um caminho que saem de x para x, com exceção do próprio arco (x, y). Analogamente, deve-se assinalar cada arco (x, y) como uma marca de herdeiro que satisfaça a regra 4, se houver pelo menos um arco sem esta marca que saem de y ou entre os arcos que compõem um caminho que chegam inversamente a partir dos arcos de entrada de y para y, exceto o próprio (x, y). Estes passos devem ser executados enquanto houver arcos que satisfaçam essas condições. Por fim, eliminam-se todos os arcos com marca de herdeiro, unindo os nós que compõem estes arcos.

Uma propriedade interessante é que os casos de teste que cobrem os arcos primitivos, também cobrem todos os outros arcos do grafo fluxo de controle (Chusho, 1987). Neste sentido, o presente trabalho verifica se é possível aplicar a mesma propriedade ao teste de mutação, ou seja, é analisar se ao matar apenas os mutantes localizados nos arcos primitivos, os mutantes localizados em outras partes do código também serão mortos.

A Figura 2.7 ilustra um exemplo dos arcos primitivos de um grafo fluxo de controle G qualquer, apresentado na Figura 2.7(a). Os arcos pontilhados, apresentados na Figura 2.7(b), são os arcos primitivos do grafo G. Ou seja, os casos de testes que cobrem os arcos $(4, 5)$ e $(4, 6)$, cobrem todos os outros arcos de G. Observe que, neste caso, a saída apresenta os arcos primitivos do grafo ao invés do grafo reduzido de herdeiros, no entanto, essa alteração será abordada com mais detalhes no Capítulo 4.3.



(a) Grafo fluxo de controle G.

(b) Arcos primitivos de G.

Figura 2.7: Exemplo de arcos primitivos.

2.5 Considerações Finais

O presente capítulo apresentou de forma sucinta os conceitos utilizados para a realização deste trabalho. Dentre os conceitos, destaca-se o teste de mutação que, embora seja considerado um critério de teste poderoso para revelar defeitos, ainda carece de melhorias – principalmente relacionadas ao custo de aplicação – para uma maior aplicação na indústria. Sendo assim, surge a necessidade de proposição de novas abordagens visando uma otimização do teste de mutação.

Um segundo conceito apresentado neste capítulo é o de arcos primitivos (ou arcos essenciais para cobertura) que combinado com teste de mutação, serviram de base para a definição da abordagem proposta neste trabalho e que será detalhada no Capítulo 4. Por fim, o presente capítulo abordou o conceito de um conjunto minimal de mutantes que representa o conjunto de mutantes necessário para preservar a adequação de um dado conjunto de teste, além de uma breve introdução de como este conceito será empregado no estudo experimental apresentado no Capítulo 4.

Capítulo 3

REVISÃO BIBLIOGRÁFICA

3.1 Considerações Iniciais

O presente capítulo apresenta os trabalhos relacionados a esta dissertação. Neste contexto, a Revisão Sistemática da Literatura (*SLR*) realizada por Pizzoleto et al. (2019), que será melhor detalhada na próxima seção, serviu como base para selecionar os estudos relacionados. Resumidamente, o trabalho de Pizzoleto et al. (2019) apresenta estudos relacionados à redução do custo do teste mutação, organizando estes estudos de acordo com a técnica utilizada. Dentre as técnicas destaca-se o uso de análise do fluxo de controle visando reduzir o custo do teste de mutação e, neste sentido, foram escolhidos os estudos que abordam o emprego desta técnica.

Ressalta-se que, além dos estudos apresentados por Pizzoleto et al. (2019), foram buscados estudos realizados posteriormente à revisão sistemática e alguns estudos que não foram selecionados, mas que abordam os conceitos apresentados (por exemplo, alguns estudos que abordam os mutantes minimais). A seguir, são apresentados os trabalhos relacionados e uma breve discussão sobre os mesmos.

3.2 Trabalhos Relacionados

No estudo apresentado por Agrawal (1994), é proposta uma técnica para selecionar um subconjunto de nós de um grafo fluxo de controle, onde os casos de teste que cobrem este conjunto reduzido de nós, cobrem todos os outros nós do grafo fluxo de controle original. Este estudo possui semelhanças ao trabalho apresentado por Chusho (1987), porém, o segundo trabalho foca nos arcos essenciais para cobertura ao invés dos nós do grafo fluxo de controle.

Como descrito anteriormente, o recente trabalho conduzido por Pizzoleto et al. (2019) apresentou uma Revisão Sistemática da Literatura com os estudos relacionados à redução do custo do teste de mutação. Os autores agruparam e sumarizaram os estudos de acordo com a técnica empregada para reduzir os custos. Ao todo, foram selecionados 153 estudos e caracterizados a partir dos objetivos empregados para redução de custo: redução no número de mutantes executados, detecção de mutantes equivalentes, reduzir o tempo de execução do teste de mutação, reduzir o número de casos de teste, evitar a criação de certos mutantes específicos (triviais e equivalentes) e geração automática de teste.

Dentre as 21 técnicas apresentadas, destaca-se o uso de análise do fluxo de controle para reduzir o custo do teste de mutação. Portanto, os estudos relacionados à esta técnica e cujo objetivo é reduzir o número de mutantes executados e mutantes equivalentes são apresentados a seguir.

No estudo realizado por Sun et al. (2017), os autores apresentaram uma abordagem para reduzir o custo do teste de mutação englobando conceitos de análise de fluxo de controle. Neste caso, os autores selecionaram um subconjunto de mutantes com base na localização do defeito, de acordo com profundidade do caminho percorrido do grafo fluxo de controle. Neste contexto, foram definidas quatro heurísticas para selecionar os mutantes e, a partir delas, quatro estratégias foram derivadas para reduzir o número de mutantes. Os resultados mostram que abordagem foi superior à mutação seletiva e randômica. De certa forma, o presente trabalho também considera a localização do defeito, entretanto, diferente de Sun et al. (2017), o foco está na geração dos mutantes considerando apenas os arcos essenciais para cobertura.

No trabalho de Just et al. (2017), os autores avaliaram se a utilidade de um mutante gerado a partir de um operador de mutação pode ser predita de acordo com o contexto do programa no qual o mutante foi gerado. Neste caso, a utilidade do mutante é avaliada de acordo com: equivalência, trivialidade e dominância. Ademais, os autores realizaram um estudo exploratório para modelar o contexto do programa pela representação da árvore sintática abstrata deste programa e, a partir de um algoritmo de aprendizado de máquina, classificar a utilidade do mutante. Os resultados mostraram que ao considerar o contexto do programa, é possível focar na geração de mutantes úteis.

Já o estudo realizado por Petrovic e Ivankovic (2017) apresentou uma abordagem para reduzir o número de mutantes que, por meio da aplicação de heurísticas, percorre a árvore sintática abstrata de um programa em teste, detectando os nós ditos improdutivos, ou seja, os nós que englobam mutantes que não serão executados.

Ji et al. (2009) apresentaram um método para *clusterizar* os mutantes por meio de análise estática, antes da geração de casos de teste e, desta forma, reduzir o número de mutantes executados. Neste caso, os autores utilizaram análise de fluxo de controle e execução simbólica para *clusterizar* os mutantes.

Patrick et al. (2012) apresentaram uma abordagem que utilizou análise do fluxo de controle e execução simbólica para reduzir o número de mutantes. Neste caso, a representação simbólica foi gerada e comparada com o programa original de acordo com a similaridade semântica para determinar os mutantes mais difíceis de serem mortos, desconsiderando os mutantes equivalentes.

Marcozzi et al. (2018) propuseram uma abordagem para reduzir o número de mutantes, focando nos mutantes equivalentes e redundantes, a partir de uma ferramenta que combina análise de fluxo de controle e validação de assertivas lógicas no código em teste.

Offutt e Pan (1997) apresentou uma abordagem englobando análise de fluxo de controle e técnicas baseadas em restrições, mostrando que as restrições de caminho auxiliam na detecção de mutantes equivalentes. Os resultados mostraram que ao utilizar está técnica, foi possível identificar, aproximadamente, 47% de mutantes equivalentes.

Harman et al. (2001) apresentaram um estudo que avaliou a relação entre análise de dependência e teste de mutação para reduzir o número de mutantes equivalentes e gerar casos de teste focando nos mutantes não equivalentes.

No trabalho de Schuler et al. (2009), os autores utilizaram análise de fluxo de controle e de dados para definir o conceito de invariante dinâmica. Estas invariantes são utilizadas para analisar as diferenças entre comportamento de mutantes em relação ao programa original, auxiliando na classificação de mutantes equivalentes e não equivalentes. Schuler e Zeller (2010) apresentaram uma abordagem que auxilia a classificação de mutantes equivalentes e não equivalentes. De forma geral, os autores abordaram o fato de que os mutantes equivalentes não têm nenhum impacto em relação à saída do programa. Ou seja, se um dado mutante causa algum impacto na estrutura interna do programa, consequentemente impactará a semântica do programa. Para avaliar o grau de impacto, os autores utilizaram análise de fluxo de controle e de dados.

No mesmo contexto, Papadakis et al. (2014) e Papadakis e Le Traon (2013) investigaram a abordagem de Schuler e Zeller (2010) e definiram uma estratégia de classificação de mutantes baseado no impacto de cobertura para reduzir mutantes equivalentes. Assim, os resultados mostraram que é possível melhorar o processo do teste de mutação, uma vez que é necessário considerar apenas um conjunto reduzido de mutantes equivalentes.

Ademais, Kintis et al. (2015) estenderam o trabalho de Schuler e Zeller (2010), relacionando o conceito do impacto na execução do programa com os mutantes de primeira ordem e no impacto da saída dos mutantes de segunda ordem para auxiliar na classificação de mutantes equivalentes.

Patel e Hierons (2016) apresentaram uma técnica para classificar mutantes equivalentes em sistemas não-determinísticos e com correção coincidente. A partir desta

técnica, foi possível atingir uma classificação de 100% de mutantes equivalentes, embora os autores tenham utilizado apenas um programa para análise.

O estudo de McMinn et al. (2019) apresentou uma abordagem que utiliza o teste de mutação no contexto de base de dados relacionais. Os autores apresentaram algoritmos que se baseiam em análise de fluxo de controle para detectar os mutantes ditos ineficazes, dentre os quais estão os mutantes equivalentes e redundantes.

A Seção 2.3.2 apresentou o conceito de um conjunto minimal de mutantes que pode ser utilizado para reduzir o conjunto de mutantes gerados, removendo os mutantes redundantes, sem perda de efetividade do conjunto de teste.

No estudo de Kurtz et al. (2016), os autores avaliaram a mutação seletiva, desconsiderando a redundância entre os mutantes. Os resultados mostraram que a mutação seletiva possui um fraco desempenho ao considerar os mutantes minimais e, desta forma, não é possível relacionar a criação dos mutantes minimais com um operador de mutação específico. Neste contexto, Delamaro et al. (2018) sugeriram que geração de mutantes minimais pode estar relacionado à sua localização no código fonte.

Kurtz et al. (2014) definiram o conceito de grafo de inclusão dos mutantes (*mutant subsumption graph*, no qual os nós raízes deste grafo compreende os mutantes não redundantes. Kurtz et al. (2015) estenderam o trabalho, mostrando como a execução simbólica pode ser utilizado na geração deste grafo.

Por fim, no recente trabalho de (Gheyi et al., 2021) apresentou uma forma de definir a relação de inclusão entre os mutantes, identificando em nível de método, os mutantes redundantes utilizando Z3 e mutação fraca.

3.3 Discussão dos Trabalhos Relacionados

A Revisão Sistemática de Pizzoleto et al. (2019) serviu como base para definir os trabalhos relacionados a esta dissertação. Foram selecionados estudos que utilizam o critério baseado em fluxo de controle visando reduzir o número de mutantes executados, além dos estudos que identificam e reduzem o número de mutantes equivalentes.

De forma geral, o trabalho que possui maior semelhança à abordagem proposta nesta dissertação é o trabalho apresentado por Sun et al. (2017), visto que ambos se baseiam na localidade dos defeitos com o objetivo de reduzir o número de mutantes executados. Porém, como descrito anteriormente, o trabalho de Sun et al. (2017) foca na profundidade da localização do defeito em um dado caminho, enquanto que a abordagem apresentada nesta dissertação foca apenas nos mutantes localizados nos arcos essenciais para cobertura.

Ainda em relação a localização do defeito, a abordagem de Petrovic e Ivankovic (2017) percorre uma árvore sintática abstrata identificando, por meio de heurísticas, os nós improdutivos que não serão utilizados na geração dos mutantes. Ao comparar com a abordagem que será apresentada no trabalho, nota-se que ambos reduzem o número de mutantes focando apenas em gerá-los em partes específicas do código. Porém, as abordagens divergem na forma como as partes do código são definidas.

Já os demais trabalhos que visam reduzir o número de mutantes não possuem grandes semelhanças com a abordagem proposta no presente trabalho, embora todos abordem o conceito de fluxo de controle. Por exemplo, Just et al. (2017) utilizaram a árvore sintática para auxiliar na classificação da utilidade dos mutantes. Ji et al. (2009) utilizaram o fluxo de controle com o objetivo de clusterizar os mutantes.

Em relação à redução dos mutantes equivalentes, é importante ressaltar que um dos objetivos do presente trabalho é verificar se ao considerar apenas os arcos primitivos de um programa, há uma redução do número de mutantes equivalentes gerados. Neste contexto, observa-se que nenhum dos trabalhos apresentados baseiam-se exclusivamente na localização do mutante equivalente. A grande maioria dos estudos focam em abordagens que utilizam análise de fluxo de controle para classificar se um dado mutante é equivalente ou não, como por exemplo: Schuler et al. (2009), Schuler e Zeller (2010), Papadakis e Le Traon (2013), Papadakis et al. (2014) e Kintis et al. (2015).

Por fim, estão alguns trabalhos que envolvem a questão dos mutantes minimais. Os trabalhos de Kurtz et al. (2014) e Kurtz et al. (2015), mostram uma forma de identificar os mutantes minimais utilizando conceitos de fluxo de controle. E a sugestão apresentada no trabalho de Delamaro et al. (2018) serviu como inspiração para um dos objetivos desta dissertação, conforme apresentado anteriormente.

3.4 Considerações Finais

Este capítulo apresentou e sintetizou os estudos relacionados a esta dissertação. Como descrito anteriormente, a Revisão Sistemática de Pizzoleto et al. (2019) serviu como base para a realização do estudo. Para isso, foram selecionados os estudos que aplicam o conceito de fluxo de controle visando reduzir o custo do teste de mutação, minimizando a quantidade de mutantes executados (ou gerados) e, além disso, estudos que utilizam fluxo de controle para reduzir e/ou identificar mutantes equivalentes.

Além disso, foram selecionados alguns estudos que se relacionam com o presente trabalho, mas que não são referenciados no trabalho de Pizzoleto et al. (2019), uma vez que os mesmos não abordam a questão da redução do custo do teste de mutação.

Capítulo 4

MUTAÇÃO EM ARCOS PRIMITIVOS: ESTUDO EXPERIMENTAL

4.1 Considerações Iniciais

Este capítulo detalha o estudo experimental realizado para avaliar e validar a abordagem proposta. A Seção 4.2 apresenta a motivação e os objetivos para realização do estudo experimental, além das questões de pesquisa. Em seguida, a Seção 4.3 apresenta a estrutura do estudo experimental e os passos executados.

4.2 Motivação e Objetivos

Conforme descrito na Seção 2.3.1, o custo do teste de mutação, advindo de um elevado número de mutantes que podem ser gerados e executados, além da dificuldade para classificação de mutantes equivalentes, são fatores que impedem uma maior utilização deste critério em situações práticas. Desta forma, é necessário prover novas abordagens visando a otimização do teste de mutação.

Neste contexto, o presente estudo tem como objetivo propor e avaliar uma forma alternativa de redução dos custos do teste de mutação ao combinar dois conceitos: **teste de mutação** e o **critério baseado em fluxo de controle**. De forma geral, o objetivo é verificar se a abordagem proposta reduz o número de mutantes executados e preserva a eficácia do conjunto de teste. Para isso, espera-se avaliar a abordagem proposta em termos de custo de aplicação e eficácia: o custo de aplicação é dado pelo número de mutantes executados, enquanto a eficácia é dada pelo escore de mutação.

Para este propósito, foram definidas as seguintes questões de pesquisa:

QP1 - Quais os ganhos, em relação ao custo de aplicação e eficácia, do uso de arcos primitivos no contexto do teste de mutação ao comparar com a mutação total (ou *full mutation*)?

QP2 - Quais os ganhos, em relação ao custo de aplicação, do uso de arcos primitivos no contexto dos mutantes equivalentes ao comparar com a mutação total (ou *full mutation*)?

QP3 - Qual a relação entre arcos primitivos e mutantes minimais?

Na primeira questão de pesquisa (*QP1*), objetivo é validar a aplicação do uso de arcos primitivos no contexto do teste de mutação, verificando se é possível reduzir o número de mutantes, mantendo a eficácia, i.e., alto escore de mutação. Já a segunda questão de pesquisa (*QP2*), avalia se é possível reduzir o número de mutantes equivalentes ao considerar apenas os mutantes gerados nos arcos primitivos. Finalmente, a terceira questão de pesquisa (*QP3*) busca verificar a relação dos mutantes minimais e os arcos primitivos. No trabalho de Delamaro et al. (2018), os autores sugerem que a geração dos mutantes minimais pode estar relacionada à sua localização no código fonte e um dos locais sugeridos são os arcos primitivos do programa. Deste modo, o presente trabalho verifica se essa suposição se mantém, ou seja, se existe uma alta concentração de mutantes minimais nos arcos primitivos de um programa em teste.

Para as responder as questões de pesquisa foi realizado um estudo experimental detalhado na Seção 4.3.

4.3 Estrutura do Estudo Experimental

Nesta seção é apresentado a estrutura do estudo experimental, especificando cada um dos passos realizados na condução do experimento. A Figura 4.1 ilustra de forma sucinta as etapas realizadas.

A primeira etapa consiste na escolha dos programas que serviram de base para a realização estudo experimental. Para isso, foram escolhidos vinte e nove programas implementados na linguagem de programação C e que foram utilizados em diversos estudos envolvendo o teste de mutação (Delamaro et al., 2014; Delamaro et al., 2014; Delamaro et al., 2014; Durelli et al., 2018). A Tabela 4.1 apresenta um panorama geral dos programas utilizados. Na primeira coluna está o nome do programa. Em seguida, tem-se o número de linhas de código (*LOC*) de cada um dos programas, que variam de 7 até 394 linhas de código, totalizando 1.400 LOC. Na terceira coluna tem-se o número de funções que compõem cada programa. Por fim, a tabela apresenta o número total de mutantes e o número total de mutantes equivalentes gerados por cada

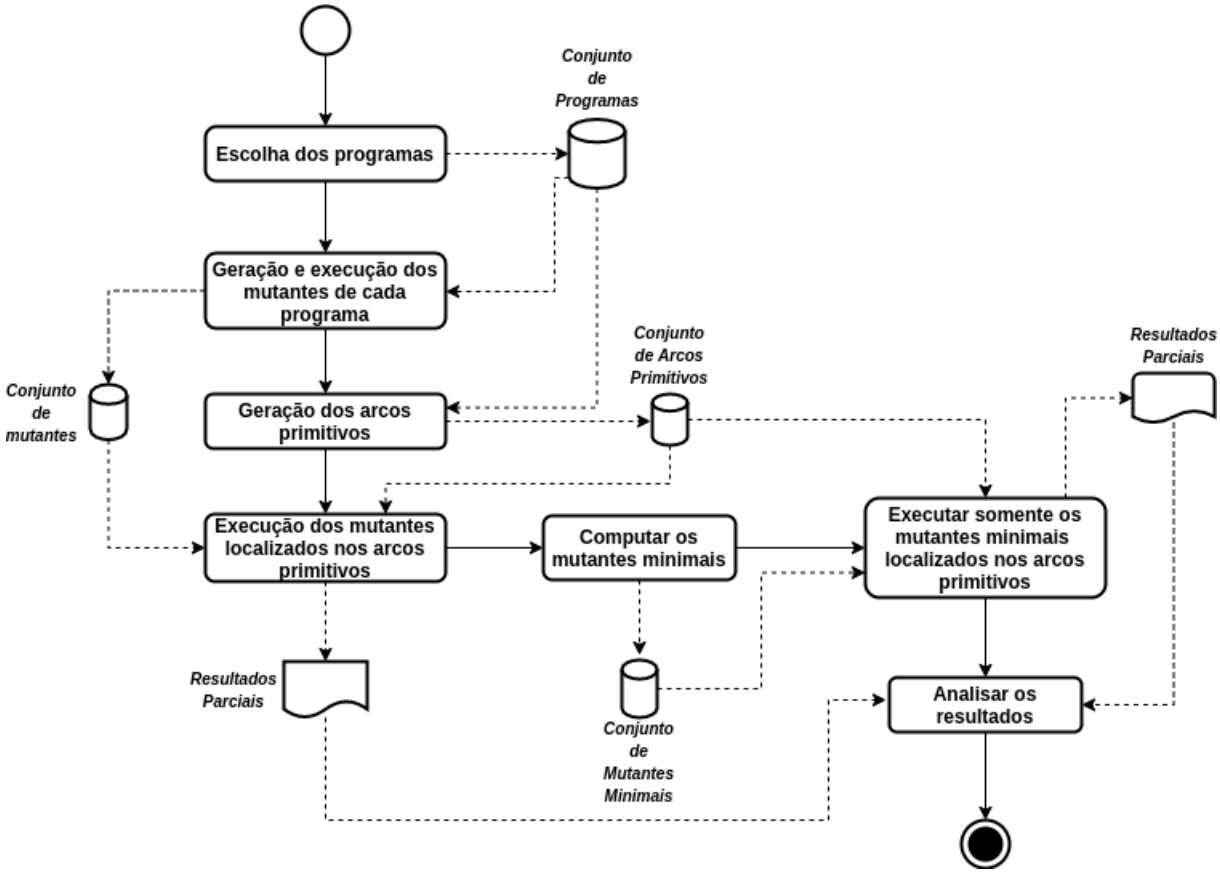


Figura 4.1: Passos realizados para a execução do experimento.

um dos programas. Ressalta-se ainda que para cada programa foi disponibilizado um conjunto adequado de teste, além dos mutantes equivalentes previamente classificados de forma manual.

Análogo ao estudo de Delamaro et al. (2014), o presente trabalho analisa cada função do programa como uma unidade, ou seja, cada função é vista como um programa independente. Desse modo, para a realização deste trabalho foram utilizadas apenas os operadores de mutação voltados para o teste de unidade. A Tabela 4.2 apresenta informações relativas às funções que foram utilizadas no estudo.

Note que o número total de funções difere na Tabela 4.1 e na Tabela 4.2. Isto acontece, pois, algumas funções geram apenas um nó na representação do grafo de fluxo de controle, impossibilitando o cálculo do conjunto de arcos primitivos da função, uma vez que para calcular os arcos primitivos é necessário que o grafo fluxo de controle contenha pelo menos um arco. Desta forma, essas funções foram descartadas da análise, gerando um conjunto de funções F, totalizando 70 funções. Houve, também, uma redução de 4,58% no número total de mutantes e de 6,55% no número total de mutantes equivalentes. Assim, as 70 funções geraram um total de 40.057 mutantes, dos quais 3.465 são mutantes equivalentes.

Ademais, é possível observar na Tabela 4.2 que as funções utilizadas possuem diferentes níveis de complexidade ciclomáticas (McCabe, 1976) – coluna *CC* – tornando

Tabela 4.1: Informações sobre os programas utilizados no estudo experimental realizado.

Programa	*LOC*	*# Funções*	*#Mutantes*	*#Equivalentes*
cal	18	1	891	77
Calculation	46	7	1.118	105
checkIt	9	1	104	3
CheckPalindrome	10	1	166	20
countPositive	9	1	151	9
date-plus	132	2	2.421	170
DigitReverser	17	1	496	43
findLast	10	1	198	8
findVal	7	1	190	10
Heap	41	7	1.079	103
InversePermutation	15	1	576	60
jday-jdate	49	2	2.821	93
lastZero	9	1	173	9
LRS	51	5	1.132	279
MergeSort	32	3	991	33
numZero	10	1	151	10
oddOrPos	9	1	361	70
pcal	204	8	6.419	1.040
power	11	1	268	6
printPrimes	35	2	715	72
quicksort	23	1	1.026	82
RecursiveSelectionSort	17	1	555	35
replace	394	20	11.101	466
sum	7	1	165	11
tcas	63	8	2384	445
testPad	24	1	629	57
trashAndTakeOut	19	2	599	27
twoPred	10	1	246	24
UnixCal	119	4	4.855	341
Total	1.400	87	41.981	3.708
Média	48,27	3	1.447,62	127,86

o conjunto de funções bastante heterogêneo. Por exemplo, algumas funções como *DefineAndRoundFraction* e *jday* (Linhas 3 e 22, respectivamente), são bastante simples e de baixa complexidade ciclomática, enquanto que as funções *printdate*, *dispatch* e *genweek* (Linhas 11, 35 e 37, respectivamente), são mais complexas e possuem uma complexidade ciclomática superior a 20.

Em seguida, foram definidas duas terminologias para auxiliar na condução dos experimentos: *nó de origem* (ou *source node*) e *nó de destino* (ou *destination node*).

Seja $G(N, E)$ um grafo fluxo de controle, onde N é o número de nós de G e E é o número de arcos de G. Seja $A \subseteq N$ o conjunto de todos os arcos primitivos de G e $(x, y) \in A$ um arco primitivo qualquer. Então, $x \in N$ é um nó de origem e $y \in N$ é um nó de destino, respectivamente, se houver um arco que sai de x em direção a y, conforme ilustrado na Figura 4.2.

Tabela 4.2: Informações sobre as funções consideradas no estudo experimental.

ID	*Programa*	*Função*	*CC*	*#M(FM)*	#Eqv(FM)
1	cal	cal	6	891	77
2	Calculation	abss	2	71	6
3		DefineAndRoundFraction	2	70	0
4		FnDivide	4	376	46
5		FnNegate	3	120	11
6		FnTimes	4	291	23
7	checkIt	checkIt	4	104	3
8	CheckPalindrome	isPalindrome	3	166	20
9	countPositive	countPositive	3	151	9
10	date-plus	incrdate	14	1.398	104
11		printdate	22	1.014	66
12	DigitReverser	digit_reverser	3	496	43
13	findLast	findLast	3	198	8
14	findVal	findVal	3	190	10
15	Heap	isSorted	3	135	7
16		less01	2	110	7
17		less02	2	42	1
18		show	2	86	8
19		sink	5	367	31
20		sort	3	191	41
21	InversePermutation	invert	7	576	60
22	jday-jdate	jday	2	828	14
23		jdate	2	1.993	79
24	lastZero	lastZero	3	173	9
25	LRS	lcp	3	204	53
26		LRS	4	331	30
27		sort	4	330	56
28	MergeSort	arraycopy	2	195	3
29		merge	2	530	14
30		mergeSort	2	266	16
31	numZero	numZero	3	151	10
32	oddOrPos	oddOrPos	5	361	70
33	pcal	byebye	2	187	6
34		dayofweek	2	347	24
35		dispatch	20	1.407	359
36		genmonth	7	1.657	266
37		genweek	25	1.998	358
38		getmmddyy	4	180	11
39		jan1	3	532	12
40	power	power	3	268	6
41	printPrimes	isDivisible	2	87	5
42		printPrimes	6	628	67
43	quicksort	quicksort	7	1.026	82
44	RecursiveSelectionSort	sort	4	555	35
45	replace	addstr	2	207	0
46		amatch	11	1.169	80
47		change	2	57	0
48		dodash	11	1.469	43
49		esc	5	541	17
50		getccl	2	530	14
51		locate	3	357	0
52		makepat	17	2.588	102
53		makesub	6	788	21
54		omatch	17	1.089	80
55		patsize	9	272	35
56		putsub	4	380	0
57		stclose	2	325	33
58		subline	6	566	0
59	sum	sum	2	165	11
60	tcas	alt_sep_test	14	1.113	162
61		Non_Crossing_Biased_Climb	6	418	127
62		Non_Crossing_Biased_Descend	6	417	125
63	testPad	pat	6	629	57
64	trashAndTakeOut	takeOut	2	280	5
65		trash	3	319	22
66	twoPred	twoPred	4	246	24
67	UnixCal	cal	9	1.536	89
68		dispatch	13	2.489	200
69		jan1	3	579	34
70		pstr	5	251	18
	Total	-	-	40.057	3.465
	Média	-	5.5	572,2	49,6

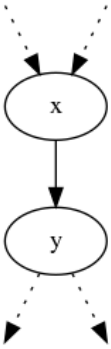


Figura 4.2: Definição dos conceitos de nó de origem e nó de destino dos arcos primitivos.

Após a definição das terminologias, foram definidos três conjuntos: S, D e $S \cup D$.

- S: conjunto de todos os mutantes localizados em um nó de origem.
- D: conjunto de todos os mutantes localizados em um nó de destino.
- $S \cup D$: união dos conjuntos S e D, ou seja, todos os mutantes localizados nos arcos primitivos.

O próximo passo consiste na geração dos mutantes para cada uma das funções F. Para realização dessa tarefa, utilizou-se a ferramenta Proteum/IM (Delamaro et al., 2000), uma ferramenta bastante utilizada para o teste de mutação em programas C. Neste estudo, foram aplicados todos os operadores de mutação de unidade da ferramenta, visto que o estudo considera o teste de mutação apenas no nível de unidade.

Após a geração dos mutantes, o conjunto de teste adequado foi executado em cada uma das funções. Por fim, a Proteum/IM gera a representação do grafo fluxo de controle da função, além do nó específico do grafo fluxo de controle no qual a mutação ocorre. Essas informações são úteis para geração do conjunto de arcos primitivos, uma vez que o grafo fluxo de controle é utilizado como entrada do algoritmo.

Em seguida, foram computados os arcos primitivos para de cada função a partir do algoritmo proposto por Chusho (1987). Chaim (1991) implementou o algoritmo (denominado *REHFLUXDA*) como um módulo da ferramenta *Poke-Tool*, porém com duas pequenas diferenças em relação ao algoritmo original:

(i) A saída do algoritmo é o conjunto de arcos primitivos ao invés do grafo reduzido de herdeiros.

(ii) A exclusão da *Regra 4* para satisfazer requisitos de fluxo de dados (Maldonado, 1991).

No presente trabalho, o módulo para computar os arcos primitivos da Poke-Tool (Chaim, 1991) foi adaptado com a implementação da *Regra 4*, objetivando satisfazer todos os requisitos do algoritmo original (Chusho, 1987). Em contrapartida, a saída do algoritmo foi mantida uma vez que a informação dos nós que compõem os arcos primitivos é essencial para realização do experimento. Note na Figura 4.3(b) que as arestas pontilhadas $(3, 4)$, $(3, 5)$, $(8, 9)$ e $(11, 12)$, são os arcos primitivos do grafo apresentado na Figura 4.3(a). Desta forma, todos os mutantes localizados nos nós 3, 8 e 11 foram adicionados no conjunto S. Analogamente, todos os mutantes localizados nos nós 4, 5, 9 e 12 foram adicionados no conjunto D.

A seguir, para cada função, os conjuntos S, D e $S \cup D$ foram determinados. No próximo passo, os casos de teste que mataram os mutantes do conjunto S foram selecionados e, tais casos de teste foram executados no conjunto total de mutantes para calcular o escore de mutação em relação à mutação total (*full mutation*). Ressalta-se que esses passos foram realizados para os conjuntos D e $S \cup D$.

Por fim, o conjunto de mutantes minimais de cada uma das funções foi determinado conforme o algoritmo proposto por Ammann et al. (2014). Análogo ao passo anterior, o escore de mutação foi computado considerando apenas os mutantes minimais localizados nos arcos primitivos, sendo que neste caso, os casos de teste que mataram os mutantes nos arcos primitivos foram executados em todos os mutantes e, a partir deste ponto, verificou-se a quantidade de mutantes minimais mortos. O escore foi calculado a partir da Equação 4.1

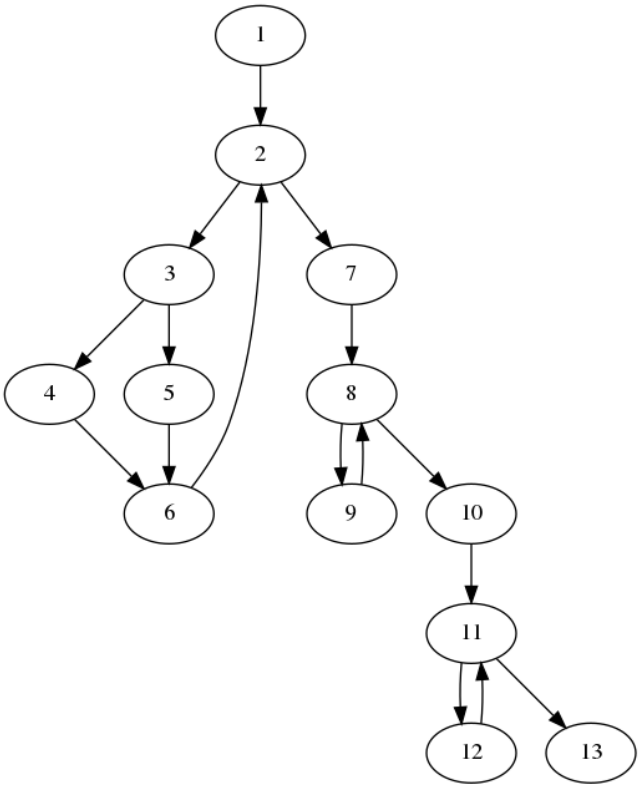
$$MS_{MIN} = \frac{\#\, Mutantes\ Minimais\ Mortos}{\#\, Total\ de\ Mutantes\ Minimais} \tag{4.1}$$

4.4 Considerações Finais

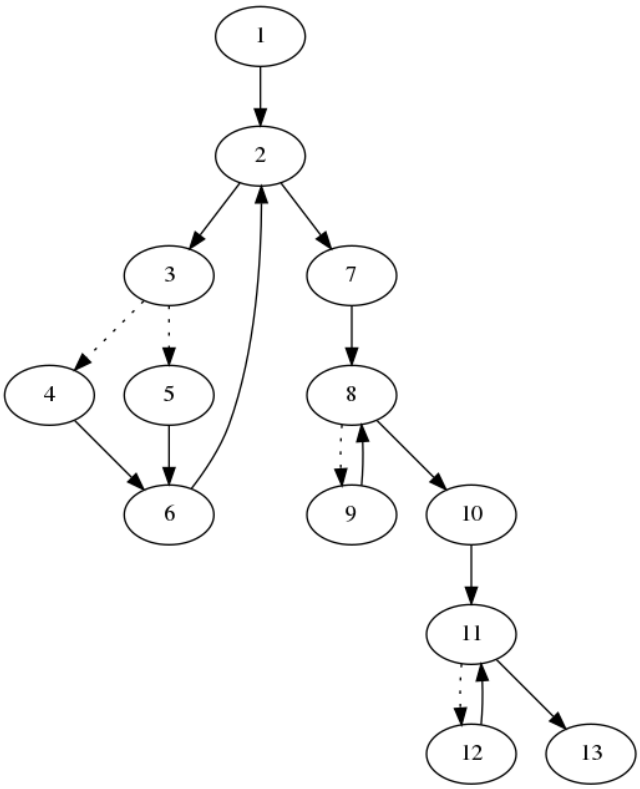
Este capítulo apresentou de forma detalhada a estrutura do estudo experimental realizado. Inicialmente, foram definidas as três questões de pesquisa que guiaram a realização deste trabalho. Em seguida, foram apresentadas uma visão geral dos programas utilizados, além das informações relativas ao número de mutantes gerados para cada programa.

É importante ressaltar que para realização do experimento, cada uma das funções que compõem os programas foi considerada como uma unidade, uma vez que, neste estudo, empregou-se o teste de mutação no nível unitário.

Por fim, são detalhados os passos realizados para a execução do experimento, bem como as ferramentas utilizadas e as adaptações necessárias para a coleta dos dados.



(a) Grafo fluxo de controle G.



(b) Arcos primitivos de G.

Figura 4.3: Exemplo de arcos primitivos utilizados no experimento.

Capítulo 5

MUTAÇÃO EM ARCOS PRIMITIVOS: RESULTADOS E ANÁLISE

5.1 Considerações Iniciais

Este capítulo apresenta os resultados do estudo experimental e uma análise dos dados obtidos. A Seção 5.2, Seção 5.3 e Seção 5.4 apresentam, respectivamente, as respostas às questões de pesquisa apresentadas na Seção 4.2. Por fim, a Seção 5.5 aborda as ameaças à validade do estudo experimental.

5.2 Mutação em Arcos Primitivos – Redução do Número de Mutantes

Na presente seção, os resultados referentes à primeira questão de pesquisa são apresentados e discutidos. Como descrito na Seção 4.2, a primeira questão de pesquisa busca evidências de que aplicar mutação apenas nos arcos primitivos é capaz de reduzir o número de mutantes que serão executados e, além disso, manter um alto escore de mutação.

Inicialmente, a Tabela 5.1 apresenta os dados relacionados à redução do número de mutantes executados. As duas primeiras colunas apresentam as informações referentes ao ID (ou linha) de cada função, seguido do nome de cada uma delas. Na coluna $\#M(FM)$, tem-se o número total de mutantes gerados por cada uma das funções. Já as colunas $\#M_{PA}(S)$, $\#M_{PA}(D)$, $\#M_{PA}(S \cup D)$ apresentam o número de mutantes que estão localizados em um nó de origem, em um nó de destino e nos arcos primitivos, respectivamente. Em seguida, nas colunas $\%Red(S)$, $\%Red(D)$, $\%Red(S \cup D)$, estão a porcentagem de redução no número de mutantes em relação ao total de mutantes

executados das funções dos conjuntos S, D e $S \cup D$, respectivamente. Para melhor visualização, a Figura 5.1 apresenta, de forma gráfica, os resultados relacionados redução do número de mutantes.

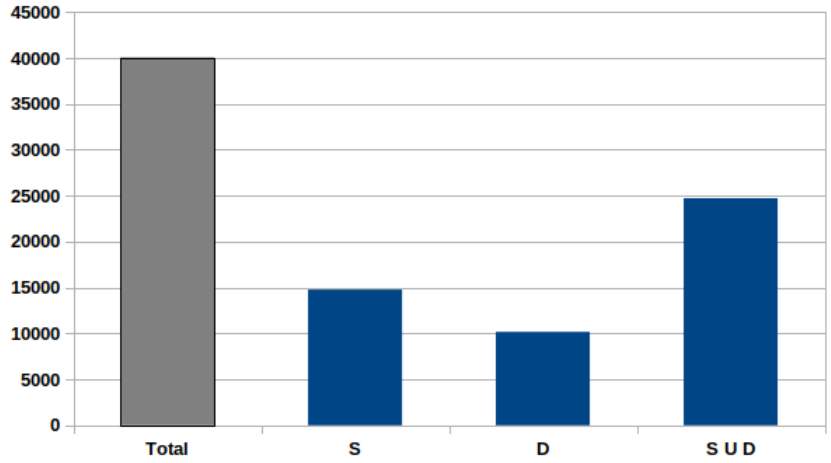


Figura 5.1: Gráfico comparativo do número total de mutantes gerados e número de mutantes nos arcos primitivos.

Analisando a Tabela 5.1, nota-se que, de um modo geral, houve redução no número de mutantes ao executar apenas àqueles localizados nos nós que compõem os arcos primitivos em comparação com o total de mutantes gerados para cada uma das funções. Observando cada conjunto individualmente, o conjunto $\#M_{PA}(S)$ executou 14.769 mutantes, gerando uma redução aproximada de 63% do número total de mutantes. Já o conjunto $\#M_{PA}(D)$ apresentou uma redução aproximada de 74% se comparado ao número total de mutantes gerados, totalizando 10.168 mutantes. E o conjunto $\#M_{PA}(S \cup D)$ obteve uma redução de aproximadamente 38%, totalizando 24.734 mutantes.

Em relação ao conjunto $\#M_{PA}(S)$, apenas 7 funções obtiveram uma porcentagem de redução do número de mutantes inferior a 20%. Por outro lado, 30 funções alcançaram uma porcentagem de redução superior a 70%. Em média, considerando apenas os mutantes localizados em um nó de origem, houve uma redução aproximada de 60% de mutantes executados por função.

Aplicando a mesma análise ao conjunto $\#M_{PA}(D)$, em 49 das 70 funções a porcentagem de redução do número de mutantes executados por função foi superior a 70%. Ademais, apenas duas funções atingiram uma porcentagem de redução inferior a 20%. Comparando $\#M_{PA}(D)$ com $\#M_{PA}(S)$, observe que o primeiro conjunto executa 31% menos mutantes do que o segundo conjunto. Uma possível explicação é que o custo computacional nos nós de origem é, em geral, superior ao custo computacional nos nós de destino. Por exemplo, em alguns casos os nós de origem possuem expressões mais complexas como laços e condicionais, enquanto que nos nós de destino existem

Tabela 5.1: Redução no número de mutantes executados.

ID	*Função*	*#M(FM)*	$\#M_{PA}(S)$	$\#M_{PA}(D)$	$\#M_{PA}(S \cup D)$	*%Red(S)*	*%Red(D)*	*%Red*$(S \cup D)$
1	cal	891	165	198	363	81,48	77,78	59,26
2	abss	71	43	28	71	39,44	60,56	0
3	DefineAndRoundFraction	70	60	10	70	14,29	85,71	0
4	FnDivide	376	161	80	241	57,18	78,72	35,9
5	FnNegate	120	35	62	97	70,83	48,33	19,17
6	FnTimes	291	137	116	253	52,92	60,14	13,06
7	checkIt	104	92	12	104	11,54	88,46	0
8	isPalindrome	166	134	21	155	19,28	87,35	6,63
9	countPositive	151	57	9	63	62,25	94,04	58,28
10	incrdate	1.398	388	49	437	72,25	96,49	68,74
11	printdate	1.014	164	679	843	83,83	33,04	16,86
12	digit_reverser	496	82	350	432	83,47	29,44	12,9
13	findLast	198	112	21	133	43,43	89,39	32,83
14	findVal	190	70	30	100	63,16	84,21	47,37
15	isSorted	135	103	8	111	23,7	94,07	17,78
16	less01	110	102	8	110	7,27	92,73	0
17	less02	42	34	8	42	19,05	80,95	0
18	show	86	46	19	65	46,51	77,91	24,42
19	sink	367	106	65	171	71,12	82,29	53,41
20	sort	191	72	57	129	62,3	70,16	32,46
21	invert	576	313	152	465	45,66	73,61	19,27
22	jday	828	68	80	148	91,79	90,34	82,13
23	jdate	1.993	1.733	75	1.808	13,05	96,24	9,28
24	lastZero	173	57	31	88	67,05	82,08	49,13
25	lcp	204	130	36	166	36,27	82,35	18,63
26	LRS	331	139	63	202	58,01	80,97	38,97
27	sort	330	57	72	129	82,73	78,18	60,91
28	arraycopy	195	48	138	186	75,38	29,23	4,62
29	merge	530	192	196	388	63,77	63,02	26,79
30	mergeSort	266	46	220	266	82,71	17,29	0
31	numZero	151	57	9	66	62,25	94,04	56,29
32	oddOrPos	361	259	9	268	28,25	97,51	25,76
33	byebye	187	21	166	187	88,77	11,23	0
34	dayofweek	347	56	149	205	83,86	57,06	40,92
35	dispatch	1.407	857	207	1.064	39,09	85,29	24,38
36	genmonth	1.657	230	388	618	86,12	76,58	62,7
37	genweek	1.998	418	1.246	1.664	79,08	37,64	16,72
38	getmmddyy	180	8	38	46	95,56	78,89	74,44
39	jan1	532	55	74	129	89,66	86,09	75,75
40	power	268	133	90	223	50,37	66,42	16,79
41	isDivisible	87	79	8	87	9,2	90,8	0
42	printPrimes	628	191	138	329	69,59	78,03	47,61
43	quicksort	1.026	372	191	563	63,74	81,38	45,13
44	sort	555	131	78	209	76,4	85,95	62,34
45	addstr	207	58	133	191	71,98	35,75	7,73
46	amatch	1.169	398	284	682	65,95	75,71	41,66
47	change	57	13	22	35	77,19	61,4	38,6
48	dodash	1.469	758	357	1.115	48,4	75,7	24,1
49	esc	541	367	155	522	32,16	71,35	3,51
50	getccl	530	139	122	261	73,77	76,98	50,75
51	locate	357	76	104	180	78,71	70,87	49,58
52	makepat	2.588	1.238	673	1.911	52,16	74	26,16
53	makesub	788	249	201	450	68,4	74,49	42,89
54	omatch	1.089	679	286	965	37,65	73,74	11,39
55	patsize	272	66	191	257	75,74	29,78	5,51
56	putsub	380	120	81	201	68,42	78,68	47,11
57	stclose	325	54	113	167	83,38	65,23	48,62
58	subline	566	117	122	239	79,33	78,45	57,77
59	sum	165	48	80	128	70,91	51,52	22,42
60	alt_sep_test	1.113	948	165	1.113	14,82	85,18	0
61	Non_Crossing_Biased_Climb	418	105	286	391	74,88	31,58	6,46
62	Non_Crossing_Biased_Descend	417	105	285	390	74,82	31,65	6,47
63	pat	629	286	56	342	54,53	91,1	45,63
64	takeOut	280	102	160	262	63,57	42,86	6,43
65	trash	319	47	114	161	85,27	64,26	49,53
66	twoPred	246	126	14	140	48,78	94,31	43,09
67	cal	1.536	244	148	392	84,11	90,36	74,48
68	dispatch	2.489	457	207	664	81,64	91,68	73,32
69	jan1	579	56	76	132	90,33	86,87	77,2
70	pstr	251	100	49	149	60,16	80,48	40,64
	Total	40.057	14.769	10.168	24.734	-	-	-
	Média	572,2	210,98	145,25	353,34	60,58	71,66	32,26

expressões mais simples como atribuição de valores a variável. Desta forma, espera-se que exista uma maior concentração de mutantes nos nós de origem.

Doravante, no conjunto $\#M_{PA}(S \cup D)$ em apenas em 6 funções a porcentagem de redução no número de mutantes foi superior a 70%, enquanto que em 27 funções a porcentagem de redução foi inferior a 20%. Em média, ao considerar apenas os mutantes localizados nos arcos primitivos, obteve-se uma redução média de 32% no número de mutantes por função.

O próximo passo consistiu na validação estatística dos resultados obtidos até o momento, a partir das hipóteses definidas e apresentas na Tabela 5.2.

Tabela 5.2: Hipóteses formalizadas – redução do número de mutantes.

Hipótese Nula	**Hipótese Alternativa**
Não existe diferença no número de mutantes entre a mutação total e a mutação nos arcos primitivos. $H1_0 : \#M(FM) = \#M_{PA}(S \cup D)$	Existe diferença no número de mutantes entre a mutação total e a mutação nos arcos primitivos. $H1_1 : \#M(FM) \neq \#M_{PA}(S \cup D)$

Inicialmente, aplicou-se o teste de normalidade Shapiro-Wilk com um nível de confiança de 95%. Os resultados mostraram que os dados não possuíam uma distribuição normal e assim, utilizou-se o teste de Wilcoxon, considerando um nível de confiança de 95% (α = 0,05). Neste caso, o teste estatístico verificou a diferença entre dois grupos: o primeiro grupo compreende o número total de mutantes gerados, enquanto o segundo grupo representa número de mutantes localizados nos nós relacionados aos arcos primitivos (coluna $\#M_{PA}(S \cup D)$ da Tabela 5.1).

O *p-value* resultante foi igual a 0,00000001138. Desta forma, como o valor foi inferior a 0,05, rejeitou-se a hipótese nula e, consequentemente, a hipótese alternativa $H1_1$ foi aceita, mostrando que o uso de arcos primitivos no contexto de teste de mutação reduz o número de mutantes que serão executados.

Embora os resultados relacionados à redução do número de mutantes sejam interessantes, não foi possível tirar nenhuma conclusão, uma vez que não é possível garantir a eficácia da estratégia. Por consequência, foi computado o escore de mutação considerando apenas os mutantes localizados nos arcos primitivos para avaliar a eficácia da estratégia em relação à mutação total. Os dados referentes ao escore de mutação são apresentados na Tabela 5.3. A coluna *Função* apresenta o nome de cada função utilizando no experimento, enquanto que as colunas $MS_{PA}(S)$, $MS_{PA}(D)$ e $MS_{PA}(S \cup D)$ apresentam o escore de mutação considerando apenas os mutantes localizados em um nó de origem, nó de destino e no arco primitivo (nó de origem e destino), respectivamente.

Conforme apresentado na Tabela 5.3, o escore de mutação médio dos conjuntos S, D e $S \cup D$ foram, respectivamente, 0,95, 0,93 e 0,98. Ao analisar cada grupo individualmente, observe que no conjunto S em 20 funções o escore de mutação foi igual à 1 ($MS_{PA}(S)$ = 1). Em 33 funções, o valor do escore de mutação está entre 0,95 e 0,99 ($0,95 \leq MS_{PA}(S) \leq 0,99$). Ademais, em apenas 5 funções o escore de mutação foi inferior a 0,8 ($MS_{PA}(S) \leq 0,8$)). Em relação ao conjunto D, em 16 funções o escore foi igual à 1, enquanto que em 27 funções o escore de mutação está entre 0,95 e 0,99 ($0,95 \leq MS_{PA}(D) \leq 0,99$). Em 10 funções o escore foi inferior a 0,8 ($MS_{PA}(D) \leq 0,8$)). Por fim, no conjunto $S \cup D$, em 42 funções o escore de mutação obtido foi igual à 1.

Tabela 5.3: Escore de mutação nos arcos primitivos – *full mutation*.

ID	*Função*	$MS_{PA}(S)$	$MS_{PA}(D)$	$MS_{PA}(S \cup D)$
1	cal	0,79	0,8	0,8
2	abss	1	0,98	1
3	DefineAndRoundFraction	1	0,95	1
4	FnDivide	1	0,94	1
5	FnNegate	1	1	1
6	FnTimes	0,98	0,99	1
7	checkIt	1	0,58	1
8	isPalindrome	0,99	0,96	0,99
9	countPositive	0,99	0,73	0,99
10	incrdate	0,92	0,79	0,92
11	printdate	0,59	0,98	1
12	digit_reverser	0,85	0,99	1
13	findLast	0,94	0,82	0,94
14	findVal	0,99	0,93	0,99
15	isSorted	1	0,78	1
16	less01	1	0,99	1
17	less02	1	1	1
18	show	0,96	0,98	1
19	sink	0,98	0,99	0,99
20	sort	0,91	0,94	0,98
21	invert	0,99	0,93	1
22	jday	0,98	0,97	0,99
23	jdate	1	0,96	1
24	lastZero	0,98	0,93	0,98
25	lcp	0,93	1	1
26	LRS	0,99	0,97	1
27	sort	0,95	0,87	0,95
28	arraycopy	0,95	0,96	1
29	merge	0,98	0,94	0,98
30	mergeSort	0,91	0,97	1
31	numZero	1	0,8	1
32	oddOrPos	0,99	0,49	0,99
33	byebye	0,91	1	1
34	dayofweek	0,79	0,99	0,99
35	dispatch	0,99	0,73	1
36	genmonth	0,94	0,94	0,94
37	genweek	0,99	0,99	0,99
38	getmmddyy	0,93	0,94	0,95
39	jan1	0,95	0,93	0,98
40	power	0,98	0,99	1
41	isDivisible	1	1	1
42	printPrimes	0,98	0,94	0,98
43	quicksort	0,99	1	1
44	sort	1	0,94	1
45	addstr	1	1	1
46	amatch	0,98	0,98	0,99
47	change	1	1	1
48	dodash	0,98	1	1
49	esc	1	0,99	1
50	getccl	0,96	1	1
51	locate	0,94	1	1
52	makepat	0,98	0,99	0,99
53	makesub	0,95	1	1
54	omatch	0,99	0,91	0,99
55	patsize	0,64	1	1
56	putsub	1	1	1
57	stclose	0,96	0,98	0,99
58	subline	1	1	1
59	sum	0,99	0,99	1
60	alt_sep_test	1	0,6	1
61	Non_Crossing_Biased_Climb	0,98	0,99	1
62	Non_Crossing_Biased_Descend	0,8	0,99	1
63	pat	0,97	0,93	0,97
64	takeOut	0,98	0,98	1
65	trash	1	0,96	1
66	twoPred	0,98	0,67	0,98
67	cal	0,98	0,96	0,98
68	dispatch	0,92	0,86	0,92
69	jan1	0,94	0,93	0,96
70	pstr	1	1	1
	Média	0,95	0,93	0,98

Em 23 funções, o valor do escore de mutação ficou entre 0,95 e 0,99. E, em apenas uma função o escore de mutação ficou abaixo de 0,90.

Analisando os dados da Tabela 5.1 e da Tabela 5.3, é possível observar que em diversos casos houve uma alta redução no número de mutantes, além de um alto escore de mutação. Por exemplo, considerando apenas os mutantes do conjunto S, as funções *sort*, *addstr*, *change*, *subline* e *trash* (Linhas 44, 45, 47, 58 e 65, respectivamente), atingiram uma redução no número de mutantes acima de 70%, além de um escore de mutação igual a 1. De forma análoga, ao considerar apenas os mutantes de D, as funções *less02*, *lcp*, *isDivisible*, *quicksort* e *pstr* (Linhas 17, 25, 41, 43 e 70, respectivamente), atingiram uma redução superior a 80% e escore de mutação igual à 1.

Em seguida, foi realizado um teste estatístico para verificar as diferenças entre o escore de mutação ao considerar a mutação total e ao considerar a mutação nos arcos primitivos. A Tabela 5.4 apresenta as hipóteses definidas.

Tabela 5.4: Hipóteses formalizadas – escore de mutação: mutação total e mutação nos arcos primitivos.

Hipótese Nula	**Hipótese Alternativa**
Não existe diferença no escore de mutação na mutação total e na mutação nos arcos primitivos. $H1_0 : MS(FM) = MS_{PA}(S \cup D)$	Existe diferença no escore de mutação na mutação total e na mutação nos arcos primitivos. $H1_1 : MS(FM) \neq MS_{PA}(S \cup D)$

O teste de normalidade Shapiro-Wilk mostrou que os dados não apresentavam uma distribuição normal e, desta forma, o teste de Wilcoxon foi aplicado com um nível de confiança de 95% (α = 0,05). Uma vez que o valor do *p-value* foi de 0,000003106, a hipótese alternativa foi aceita. Ou seja, embora o escore médio final da mutação nos arcos primitivos tenha sido próximo à 1, constata-se uma diferença estatística do escore de mutação obtido na abordagem dos arcos primitivos e na mutação total.

Um ponto importante a ser ressaltado é que o escore final obtido ao considerar todo o conjunto de mutantes pode não refletir a real adequação do conjunto de teste, visto que este de escore pode estar ”inflado” pelos mutantes redundantes. Desta forma, torna-se necessário computar o escore considerando apenas os mutantes minimais. Os resultados referentes a esta análise são apresentados na Seção 5.4.

5.3 Mutação em Arcos Primitivos – Redução de Mutantes Equivalentes

A seção anterior trouxe uma análise sobre a utilização do conceito de arcos primitivos no teste de mutação mostrando que, no geral, é possível reduzir o número de mutantes executados e ainda preservar um alto escore de mutação. Agora, o objetivo é verificar se o uso de arcos primitivos no contexto do teste de mutação também reduz o número de mutantes equivalentes.

A Tabela 5.5 apresenta os resultados relacionados à redução dos mutantes equivalentes. Na coluna $Eqv(FM)$ tem-se o número total de mutantes equivalentes gerado para cada uma das funções. Já as colunas $\#Eqv_{PA}(S)$, $\#Eqv_{PA}(D)$ e $\#Eqv_{PA}(S \cup D)$, apresentam o número de mutantes equivalentes nos conjuntos S, D e $S \cup D$, respectivamente. Por fim, as colunas $\%Red_{Eqv}(S)$, $\%Red_{Eqv}(D)$ e $\%Red_{Eqv}(S \cup D)$ informam a porcentagem de redução do número de mutantes equivalentes dos conjuntos S, D e $S \cup D$, respectivamente, em relação ao número total de mutantes equivalentes gerados.

Analisando os resultados, a média de redução do número de mutantes equivalentes dos conjuntos S, D e $S \cup D$ para cada função é de 59%, 63% e 31%, respectivamente. Note que, embora o número de mutantes localizados nos nós de destino seja superior ao número de mutantes localizados nos nós de origem, a média de redução é maior em D do que em S. A Figura 5.2 ilustra essa diferença entre o número total de mutantes equivalentes gerados em relação ao número de mutantes equivalentes nos arcos primitivos, considerando cada um dos três conjuntos.

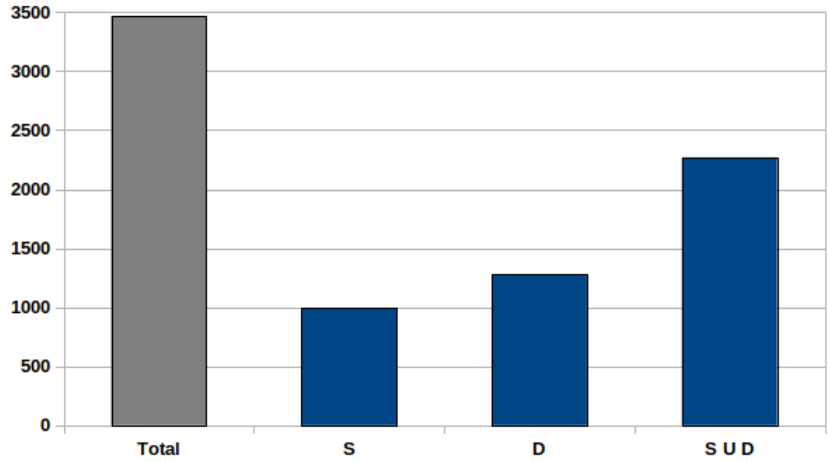


Figura 5.2: Gráfico comparativo do número total de mutantes equivalentes gerados e número de mutantes equivalentes nos arcos primitivos.

No conjunto S, em 9 funções a porcentagem de redução foi de 100%, ou seja, não geraram nenhum mutante equivalente. Porém, note que 6 dessas 9 funções (*less02*,

Tabela 5.5: Redução no número de mutantes equivalentes.

ID	*Função*	*#Eqv(FM)*	$\#Eqv_{PA}(S)$	$\#Eqv_{PA}(D)$	$\#Eqv_{PA}(S \cup D)$	*%Red(S)*	*%Red(D)*	*%Red($S \cup D$)*
1	cal	77	9	13	22	88,31	83,12	71,43
2	abss	6	2	4	6	66,67	33,33	0
3	DefineAndRoundFraction	0	0	0	0	0	0	0
4	FnDivide	46	14	3	17	69,57	93,48	63,04
5	FnNegate	11	9	2	11	18,18	81,82	0
6	FnTimes	23	22	1	23	4,35	95,65	0
7	checkIt	3	3	0	3	0	100	0
8	isPalindrome	20	19	0	19	5	100	5
9	countPositive	9	1	0	1	88,89	100	88,89
10	incrdate	104	46	0	46	55,77	100	55,77
11	printdate	66	16	34	50	75,76	48,48	24,24
12	digit_reverser	43	24	13	37	44,19	69,77	13,95
13	findLast	8	2	2	4	75	75	50
14	findVal	10	1	2	3	90	80	70
15	isSorted	7	7	0	7	0	100	0
16	less01	7	6	1	7	14,29	85,71	0
17	less02	1	0	1	1	100	0	0
18	show	8	5	1	6	37,5	87,5	25
19	sink	31	8	7	15	74,19	77,42	51,61
20	sort	41	17	6	23	58,54	85,37	43,9
21	invert	60	32	16	48	46,67	73,33	20
22	jday	14	2	0	2	85,71	100	85,71
23	jdate	79	65	3	68	17,72	96,2	13,92
24	lastZero	9	1	1	2	88,89	88,89	77,78
25	lcp	53	30	9	39	43,4	83,02	26,42
26	LRS	30	10	5	15	66,67	83,33	50
27	sort	56	5	6	11	91,07	89,29	80,36
28	arraycopy	3	1	2	3	66,67	33,33	0
29	merge	14	10	0	10	28,57	100	28,57
30	mergeSort	16	0	16	16	100	0	0
31	numZero	10	1	0	1	90	100	90
32	oddOrPos	70	59	0	59	15,71	100	15,71
33	byebye	6	6	0	6	0	100	0
34	dayofweek	24	5	8	13	79,17	66,67	45,83
35	dispatch	359	165	165	330	54,04	54,04	8,08
36	genmonth	266	32	70	102	87,97	73,68	61,65
37	genweek	358	55	214	269	84,64	40,22	24,86
38	getmmddyy	11	0	2	2	100	81,82	81,82
39	jan1	12	0	1	1	100	91,67	91,67
40	power	6	6	0	6	0	100	0
41	isDivisible	5	5	0	5	0	100	0
42	printPrimes	67	17	13	30	74,63	80,6	55,22
43	quicksort	82	28	8	36	65,85	90,24	56,1
44	sort	35	9	4	13	74,29	88,57	62,86
45	addstr	0	0	0	0	0	0	0
46	amatch	80	17	15	32	78,75	81,25	60
47	change	0	0	0	0	0	0	0
48	dodash	43	0	43	43	100	0	0
49	esc	17	4	13	17	76,47	23,53	0
50	getccl	14	0	14	14	100	0	0
51	locate	0	0	0	0	0	0	0
52	makepat	102	3	85	88	97,06	16,67	13,73
53	makesub	21	0	21	21	100	0	0
54	omatch	80	24	43	67	70	46,25	16,25
55	patsize	35	0	35	35	100	0	0
56	putsub	0	0	0	0	0	0	0
57	stclose	33	3	12	15	90,91	63,64	54,55
58	subline	0	0	0	0	0	0	0
59	sum	11	8	1	9	27,27	90,91	18,18
60	alt_sep_test	162	95	67	162	41,36	58,64	0
61	Non_Crossing_Biased_Climb	127	3	122	125	97,64	3,94	1,57
62	Non_Crossing_Biased_Descend	125	2	121	123	98,4	3,2	1,6
63	pat	57	18	5	23	68,42	91,23	59,65
64	takeOut	5	1	3	4	80	40	20
65	trash	22	0	5	5	100	77,27	77,27
66	twoPred	24	3	0	3	87,5	100	87,5
67	cal	89	17	8	25	80,9	91,01	71,91
68	dispatch	200	29	22	51	85,5	89	74,5
69	jan1	34	3	6	9	91,18	82,35	73,53
70	pstr	18	5	5	10	72,22	72,22	44,44
	Total	3.465	990	1.279	2.269	-	-	-
	Média	49,6	14,14	18,27	32,41	59,59	63,47	31,63

mergeSort, *dodash*, *getcch*, *makesub* e *patsize*), não geraram mutantes equivalentes nos nós de origem, porém todos os mutantes equivalentes estavam localizados em um nó de destino. 3 das 9 funções (*getmmddyy*, *jan1*[1] e *trash*) não geram mutantes equivalentes no nó de origem e reduziram de forma significativa o número de mutantes equivalentes nos nós de destinos. Ademais, em 37 funções a redução foi superior a 70% enquanto que em 20 funções a redução foi inferior a 20%. Finalmente, 10 Funções não

[1] Função jan1 do programa pcal, Linha 69

reduziram o número de mutantes equivalentes, ou seja, todos os mutantes equivalentes estavam localizados em um nó de origem.

No conjunto D, 13 funções não geraram mutantes equivalentes. Em 5 das 13 funções (*checkIt*, *isSorted*, *byebye*, *power* e *isDivisible*), todos os mutantes equivalentes estavam localizados em um nó de origem. Por outro lado, 8 funções não geraram mutantes equivalentes nos nós de destino, além de reduzi-los significativamente nos nós de origem. Ressalta-se que 42 funções atingiram a porcentagem de redução superior a 70%, em 15 funções a porcentagem de redução foi inferior a 20% e em 12 funções, todos os mutantes equivalentes estavam localizados nos nós de destino.

Ao considerar o conjunto $S \cup D$, note que em 24 funções não houve redução no número de mutantes equivalentes. Em 14 funções a porcentagem de redução foi superior a 70%. E, finalmente, nenhuma das 70 funções atingiram 100% de redução.

O próximo passo é verificar estatisticamente se a aplicação de arcos primitivos no teste de mutação reduz o número de mutantes equivalentes, conforme as hipóteses definidas na Tabela 5.6.

Tabela 5.6: Hipóteses formalizadas – redução do número de mutantes equivalentes.

Hipótese Nula	**Hipótese Alternativa**
Não existe diferença no número de mutantes equivalentes entre a mutação total e a mutação nos arcos primitivos. $H1_0 : \#Eqv_{PA}(FM) = \#Eqv_{PA}(S \cup D)$	Existe diferença no número de mutantes equivalentes entre a mutação total e a mutação nos arcos primitivos. $H1_1 : \#Eqv_{PA} \neq \#Eqv_{PA}(S \cup D)$

Inicialmente, o teste de normalidade Shapiro-Wilk foi aplicado com um nível de confiança de 95%. Uma vez que os dados não possuíam uma distribuição normal, aplicou-se teste de Wilcoxon, utilizando um nível de confiança de 95% (α = 0,05). O *p-value* resultante foi igual à 0,000000003613 e, uma vez que o valor é inferior a 0,05, a hipótese alternativa foi aceita, demonstrando que é possível reduzir o número de mutantes equivalentes, utilizando o conceito de arcos primitivos no contexto do teste de mutação.

5.4 Mutação em Arcos Primitivos – Relação entre Mutantes Minimais e Arcos Primitivos

Nesta seção estão os resultados relacionados aos mutantes minimais. De forma geral, o objetivo é verificar se existe a relação entre os mutantes minimais e os arcos primitivos, ou seja, se os arcos primitivos tendem a gerar um maior número de mutantes minimais (Delamaro et al., 2018).

O primeiro passo é verificar se existe uma concentração de mutantes minimais nos nós que compõem os arcos primitivos. Os resultados desta análise são apresentados na Tabela 5.7. A coluna $\#Min(FM)$ apresenta o total de mutantes minimais gerados para cada uma das funções. As colunas $\#Min_{PA}(S)$, $\#Min_{PA}(D)$ e $\#Min_{PA}(S \cup D)$, apresenta o número de mutantes minimais gerados para os conjuntos S, D e $S \cup D$, respectivamente.

Observando os resultados, é possível verificar que dos 479 mutantes minimais gerados para todas as funções, 336 estão em um dos nós que compõem os arcos primitivos, representando 70% do total de mutantes minimais. Analisando cada conjunto de forma individual, note que o conjunto S concentra 177 mutantes minimais, com uma média de 2,52 mutantes por função. Analogamente, o conjunto D concentra 159 mutantes minimais, com uma média de 2,27 mutantes por função.

O próximo passo é validar estatisticamente os resultados. Para isso, foram definidas as hipóteses apresentadas na Tabela 5.8.

Novamente, os dados não apresentaram uma distribuição normal e, portanto, o teste de Wilcoxon foi aplicado com nível de confiança de 95% (α = 0,05). Como *p-value* resultante (0,00000000636) é inferior a 0,05, pode-se concluir que, de fato, existe uma alta concentração de mutantes minimais nos arcos primitivos.

A Seção 5.2, apresentou os dados relacionados ao escore de mutação ao considerar o conjunto total de mutantes. Como destacado, o escore apresentado pode não refletir a real adequação do conjunto de teste, uma vez que os mutantes redundantes podem gerar uma imprecisão no escore. Por esse motivo, é interessante computar o escore de mutação considerando apenas os mutantes minimais. A Tabela 5.9 apresenta os resultados obtidos em cada um dos conjuntos S, D e $S \cup D$.

Ao avaliar os resultados, nota-se que houve uma redução considerável no escore médio de cada conjunto, conforme aprestado no gráfico da Figura 5.3. Observe que o escore de mutação médio do conjunto S reduziu de 0,95 para 0,69. No conjunto D, o escore médio reduziu de 0,93 para 0,58. E no conjunto $S \cup D$ o escore médio reduziu de 0,98 para 0,86.

Analisando o conjunto S de forma individual, em 20 funções o escore de mutação manteve-se em 1, ou seja, todos os mutantes minimais em S foram mortos. Por outro lado, em duas funções (*dayofweek* e *locate*) o escore de mutação foi igual à 0, logo, todos os mutantes minimais permaneceram vivos. Ademais, apenas uma função (*omatch*) o escore de mutação esteve entre 0,90 e 0,99. Por fim, desconsiderando as funções cujo escore foi igual à 0, tem-se que 10 funções obtiveram uma redução no escore superior a 50%.

De forma análoga, no conjunto D, em 16 funções o escore de mutação considerando apenas os mutantes minimais manteve-se igual à 1. Em 4 funções (*isSorted*, *less01*, *oddOrPos* e *getmmddyy*) o escore foi igual à 0, mostrando que todos os mutantes

Tabela 5.7: Número de mutantes minimais nos arcos primitivos.

ID	*Função*	*#Min(FM)*	$Min_{PA}(S)$	$Min_{PA}(D)$	$Min_{PA}(S \cup D)$
1	cal	7	1	1	2
2	abss	8	2	6	8
3	DefineAndRoundFraction	3	3	0	3
4	FnDivide	5	3	2	5
5	FnNegate	1	0	0	0
6	FnTimes	4	1	3	4
7	checkIt	7	7	0	7
8	isPalindrome	6	4	1	5
9	countPositive	5	4	0	4
10	incrdate	14	3	1	4
11	printdate	21	2	19	21
12	digit_reverser	4	0	2	2
13	findLast	5	2	0	2
14	findVal	5	2	1	3
15	isSorted	4	4	0	4
16	less01	1	1	0	1
17	less02	3	2	1	3
18	show	3	1	1	2
19	sink	4	1	1	2
20	sort	4	0	1	1
21	invert	7	4	2	6
22	jday	8	2	1	3
23	jdate	7	7	0	7
24	lastZero	4	2	1	3
25	lcp	2	1	1	2
26	LRS	5	2	1	3
27	sort	6	0	0	0
28	arraycopy	5	2	3	5
29	merge	7	2	1	3
30	mergeSort	4	2	2	4
31	numZero	3	1	0	1
32	oddOrPos	5	4	0	4
33	byebye	4	1	3	4
34	dayofweek	5	0	3	3
35	dispatch	9	7	2	9
36	genmonth	6	0	1	1
37	genweek	6	1	1	2
38	getmmddyy	3	1	0	1
39	jan1	8	2	1	3
40	power	5	1	4	5
41	isDivisible	1	0	1	1
42	printPrimes	3	0	0	0
43	quicksort	5	2	2	4
44	sort	7	3	1	4
45	addstr	10	4	6	10
46	amatch	8	2	2	4
47	change	2	0	1	1
48	dodash	6	1	5	6
49	esc	5	3	2	5
50	getccl	8	0	6	6
51	locate	1	0	1	1
52	makepat	11	5	5	10
53	makesub	2	0	2	2
54	omatch	28	24	3	27
55	patsize	25	0	24	24
56	putsub	1	0	0	0
57	stclose	19	2	8	10
58	subline	1	0	0	0
59	sum	4	2	1	3
60	alt_sep_test	26	26	0	26
61	Non_Crossing_Biased_Climb	10	2	8	10
62	Non_Crossing_Biased_Descend	10	3	7	10
63	pat	8	2	1	3
64	takeOut	6	2	4	6
65	trash	5	1	1	2
66	twoPred	8	2	0	2
67	cal	6	0	0	0
68	dispatch	18	5	0	5
69	jan1	7	1	1	2
70	pstr	5	0	0	0
	Total	479	177	159	336
	Média	6,84	2,52	2,27	4,8

Tabela 5.8: Hipóteses formalizadas – concentração de mutantes minimais.

Hipótese Nula	Hipótese Alternativa
Não existe uma concentração de mutantes minimais nos arcos primitivos. $H1_0$: $\#Min(FM) \neq \#Min_{PA}(S \cup D)$	Existe uma concentração de mutantes minimais nos arcos primitivos. $H1_1$: $\#Min(FM) = \#Min_{PA}(S \cup D)$

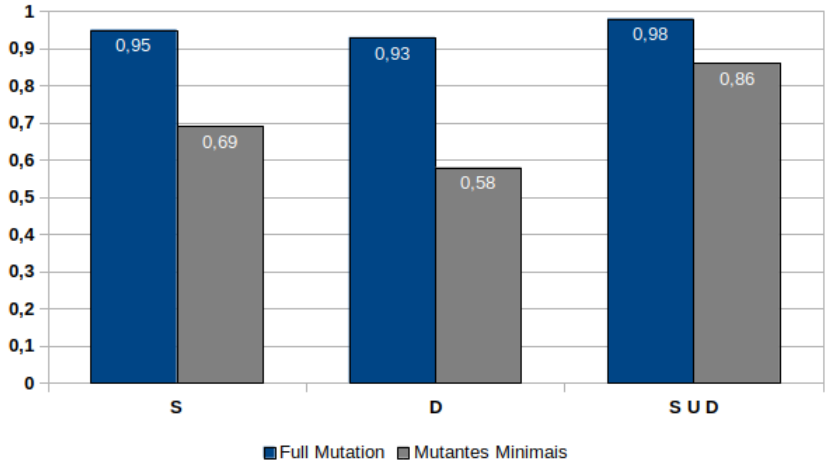


Figura 5.3: Gráfico comparativo do escore de mutação médio: *full mutation* x mutantes minimais.

minimais permaneceram vivos. Apenas duas funções atingiram um escore entre 0,90 e 0,99. Finalmente, 23 funções alcançaram uma redução no escore de mutação superior a 50%. Por exemplo, a função *dispatch* (Linha 68), reduziu o escore de 0,89 para 0,11, totalizando 87%.

Por fim, no conjunto $S \cup D$, 42 funções alcançaram o escore de 1 e em três funções o escore atingiu um valor entre 0,90 e 0,99. Diferente dos conjuntos S e D, em nenhuma das funções o escore foi igual a zero. Ademais, apenas 7 funções tiveram uma redução no escore acima de 50%.

Para finalizar, foi realizado um teste estatístico para verificar as diferenças entre o escore da mutação nos arcos primitivos e o escore de mutação nos arcos primitivos ao considerar apenas os mutantes minimais. A Tabela 5.10 apresenta as hipóteses definidas.

Como os dados não apresentavam uma distribuição normal, o teste de Wilcoxon foi aplicado com nível de confiança de 95%. O *p-value* resultante obtido foi de 0,000003949 e, desta forma, concluiu-se que existe uma diferença no escore de mutação, mostrando que no caso da mutação nos arcos primitivos ao considerar a mutação total, o escore resultante estava ”inflado” pelos mutantes redundantes.

Embora a média do escore de mutação para os conjuntos tenham sido relativamente baixos para os conjuntos S e D, é importante considerar que, em alguns casos, a baixa cardinalidade do conjunto de mutantes minimais para uma função pode influenciar numa diminuição do escore. Além disso, é sempre importante fazer a ressalva que existe um custo atrelado ao cálculo dos mutantes minimais, uma vez que é necessário

Tabela 5.9: Escore de mutação considerando apenas os mutantes minimais.

ID	*Função*	$MS_{MIN}(S)$	$MS_{MIN}(D)$	$MS_{MIN}(S \cup D)$
1	cal	0,43	0,57	0,57
2	abss	1	0,88	1
3	DefineAndRoundFraction	1	0,33	1
4	FnDivide	1	0,4	1
5	FnNegate	1	1	1
6	FnTimes	0,5	0,75	1
7	checkIt	1	0,14	1
8	isPalindrome	0,83	0,67	0,83
9	countPositive	0,8	0,2	0,8
10	incrdate	0,57	0,14	0,64
11	printdate	0,14	0,95	1
12	digit_reverser	0,5	0,75	1
13	findLast	0,6	0,2	0,6
14	findVal	0,8	0,4	0,8
15	isSorted	1	0	1
16	less01	1	0	1
17	less02	1	1	1
18	show	0,67	0,67	1
19	sink	0,5	0,75	0,75
20	sort	0,5	0,5	0,75
21	invert	0,86	0,43	1
22	jday	0,38	0,25	0,5
23	jdate	1	0,29	1
24	lastZero	0,5	0,25	0,75
25	lcp	0,5	1	1
26	LRS	0,8	0,6	1
27	sort	0,33	0,17	0,33
28	arraycopy	0,6	0,6	1
29	merge	0,71	0,29	0,71
30	mergeSort	0,75	0,5	1
31	numZero	1	0,33	1
32	oddOrPos	0,8	0	0,8
33	byebye	0,5	1	1
34	dayofweek	0	0,8	0,8
35	dispatch	0,89	0,56	1
36	genmonth	0,33	0,17	0,33
37	genweek	0,83	0,83	0,83
38	getmmddyy	0,33	0	0,33
39	jan1	0,75	0,25	0,75
40	power	0,8	0,8	1
41	isDivisible	1	1	1
42	printPrimes	0,33	0,33	0,33
43	quicksort	0,8	1	1
44	sort	1	0,29	1
45	addstr	1	1	1
46	amatch	0,5	0,5	0,63
47	change	1	1	1
48	dodash	0,67	1	1
49	esc	1	0,6	1
50	getccl	0,5	1	1
51	locate	0	1	1
52	makepat	0,73	0,55	0,91
53	makesub	0,5	1	1
54	omatch	0,96	0,68	0,96
55	patsize	0,2	1	1
56	putsub	1	1	1
57	stclose	0,68	0,84	0,95
58	subline	1	1	1
59	sum	0,75	0,75	1
60	alt_sep_test	1	0,19	1
61	Non_Crossing_Biased_Climb	0,7	0,9	1
62	Non_Crossing_Biased_Descend	0,6	0,9	1
63	pat	0,63	0,25	0,63
64	takeOut	0,5	0,67	1
65	trash	1	0,6	1
66	twoPred	0,88	0,25	0,88
67	cal	0,33	0,17	0,33
68	dispatch	0,39	0,11	0,39
69	jan1	0,29	0,29	0,43
70	pstr	1	1	1
	Média	0,69	0,58	0,86

Tabela 5.10: Hipóteses formalizadas – escore de mutação: mutação nos arcos primitivos e mutação nos arcos primitivos considerando apenas os mutantes minimais.

Hipótese Nula	Hipótese Alternativa
Não existe diferença no escore de mutação na mutação nos arcos primitivos e na mutação nos arcos primitivos considerando apenas os mutantes minimais. $H1_0 : MS_{PA}(S \cup D) = MS_{MIN}(S \cup D)$	Existe diferença no escore de mutação na mutação nos arcos primitivos e na mutação nos arcos primitivos considerando apenas os mutantes minimais. $H1_1 : MS_{PA}(S \cup D) \neq MS_{MIN}(S \cup D)$

executar ao menos uma vez, todos os casos de teste em todos os mutantes gerados. Ou seja, é necessário realizar ao menos um *full mutation* para computar o conjunto de mutantes minimais.

5.5 Ameaças à Validade

Os estudos experimentais estão sujeitos a ameaças que, de certa forma, podem invalidá-los. Por conseguinte, é necessário apresentar e discutir tais ameaças. A seguir, são apresentas as ameaças à validade do estudo experimental realizado

5.5.1 Validade Interna

Em relação à validade interna, a principal ameaça está relacionada a implementação do algoritmo para computar os arcos primitivos. Para minimizar os riscos e apresentar um resultado preciso, o algoritmo que computa arcos primitivos da ferramenta Poke-Tool (Chaim, 1991) foi adaptado para satisfazer os requisitos do algoritmo original (Chusho, 1987), conforme detalhado na Seção 4.3. Ademais, os arcos primitivos gerados foram checados manualmente para que não houvesse imprecisão nos resultados.

Uma segunda ameaça à validade interna está relacionada a geração dos mutantes. No experimento, os mutantes foram gerados e executados a partir da Proteum/IM (Delamaro et al., 2000), que é uma ferramenta de teste de mutação para programas C, utilizada em diversos estudos envolvendo mutação.

Por fim, a forma de coleta de dados também é uma ameaça a validade interna. Foram implementados scripts em Python para automatizar essa atividade. Ressalta-se que todos os scripts foram revisados e checados manualmente para não gerar resultados imprecisos ou incorretos.

5.5.2 Validade Externa

A ameaça à validade externa está relacionada ao conjunto de programas utilizados no experimento, uma vez que os programas podem não ser representativo. Para isso, foram selecionados um conjunto de programas utilizados em outros estudos envolvendo teste de mutação. Além disso, conforme descrito na Seção 4.3, cada uma das funções dos programas foi considerada como uma unidade, visto que o teste de mutação é um critério de teste unitário. Desta forma, para realização do experimento utilizou-se um conjunto heterogêneo de funções.

Porém, os resultados podem não ser generalizados para todos os programas, uma vez que programas mais complexos e de outros domínios, como por exemplo, programas da indústria, podem produzir resultados diferentes.

Por fim, este experimento foi realizado utilizando apenas um conjunto adequado de teste. Diferentes conjuntos de testes podem acarretar em diferentes resultados.

5.5.3 Validade de Construção

A ameaça à validade de construção do experimento está relacionada a forma de avaliação da estratégia. Para isso, utilizou-se o escore de mutação, que também é empregado em diversos estudos envolvendo mutação.

5.5.4 Validade de Conclusão

A ameaça à validade de conclusão está relacionada a forma de obter as conclusões do experimento a partir dos resultados obtidos. Para isso, os resultados foram avaliados estatisticamente, buscando reduzir ao máximo os riscos desta ameaça.

5.6 Considerações Finais

O presente capítulo apresentou os resultados do experimento e as respostas às questões de pesquisa apresentada no capítulo anterior. Foi possível observar que, no geral, os resultados foram interessantes.

Em relação à primeira questão de pesquisa, os resultados mostraram que foi possível obter uma boa redução no número de mutantes executados e um alto escore de mutação. Além disso, testes estatísticos foram realizados para aumentar a confiança sobre os resultados obtidos. Em relação à segunda questão de pesquisa, os

dados mostraram que a abordagem proposta também reduz o número de mutantes equivalentes.

E os resultados relacionados a terceira questão de pesquisa mostraram que existe uma alta concentração de mutantes minimais. Além disso, foi computado o escore de mutação considerando apenas os mutantes minimais com o objetivo de verificar a eficácia da estratégia proposta, ao desconsiderar os mutantes redundantes.

Para finalizar, o presente capítulo apresentou as possíveis ameaças a validade que podem invalidar os resultados e as conclusões obtidas do estudo experimental.

Capítulo 6

COMPARAÇÃO: MUTAÇÃO NOS ARCOS PRIMITIVOS X MUTAÇÃO ALEATÓRIA

6.1 Considerações Iniciais

O Capítulo 5 apresentou os resultados relacionados ao uso dos arcos primitivos no contexto do teste de mutação. Agora, o objetivo é comparar a abordagem proposta com a mutação aleatória, que é uma abordagem já existente e é utilizada como *baseline* para avaliar estratégias de redução do custo do teste de mutação (Gopinath et al., 2017). A Seção 6.2 detalha os passos realizados para a comparação das estratégias. A Seção 6.3 apresenta e analisa os resultados obtidos.

6.2 Estrutura para Realização da Comparação entre as Abordagens

Inicialmente, para realizar a comparação da abordagem proposta neste trabalho com a mutação aleatória, foram necessários realizar algumas etapas, conforme descrito a seguir.

Seja F o conjunto de todas as funções e $f_i \in F, 1 \leq i \leq 70$, uma função qualquer. Para cada função f_i, selecionou-se um conjunto aleatório de mutantes M_i, tal que a cardinalidade do conjunto M_i fosse igual a cardinalidade do conjunto de mutantes localizados nos arcos primitivos, ou seja, $|M_i| = |M_{PA}(f_i)|$. Por exemplo, a função *cal* apresentado na Linha 1 da Tabela 5.1, totalizou 363 mutantes nos arcos primitivos. Logo, foram selecionados aleatoriamente 363 mutantes para esta função ($|M_{cal} = 363|$).

Em seguida, o escore de mutação do conjunto aleatório de mutantes foi computado considerando apenas os mutantes minimais, para evitar que os mutantes redundantes gerassem uma imprecisão no escore de mutação. Para apresentar um escore de mutação mais preciso e sem a interferência de *outliers*, os passos anteriores foram repetidos por 10 vezes para cada função e o escore final foi dado pela média do escore das 10 execuções.

6.3 Resultados e Análises

A presente seção, apresenta os dados e a análise referentes a comparação da abordagem proposta com a mutação aleatória. A Tabela 6.1 apresenta os resultados obtidos. A coluna $MS_{MIN}(S \cup D)$ apresenta o escore de mutação das funções ao utilizar a abordagem dos arcos primitivos conforma apresentado na Tabela 5.9. Já a coluna $MS_{Random}(S \cup D)$ apresenta a média do escore de mutação do conjunto aleatório de mutantes, após as 10 execuções, conforme apresentado na seção anterior.

É possível observar que o escore de mutação médio das funções foi um pouco superior na mutação aleatória (0,90 > 0,86). Isso ocorre, pois em alguns casos, existe uma grande diferença entre os escores. Por exemplo, na função *genmonth* (Linha 36), o escore obtido utilizando a abordagem dos arcos primitivos foi de 0,33 enquanto o escore obtido na mutação aleatória foi de 0,95. Esta constatação pode ser observada no diagrama de caixas apresentado na Figura 6.1 e na Tabela 6.2.

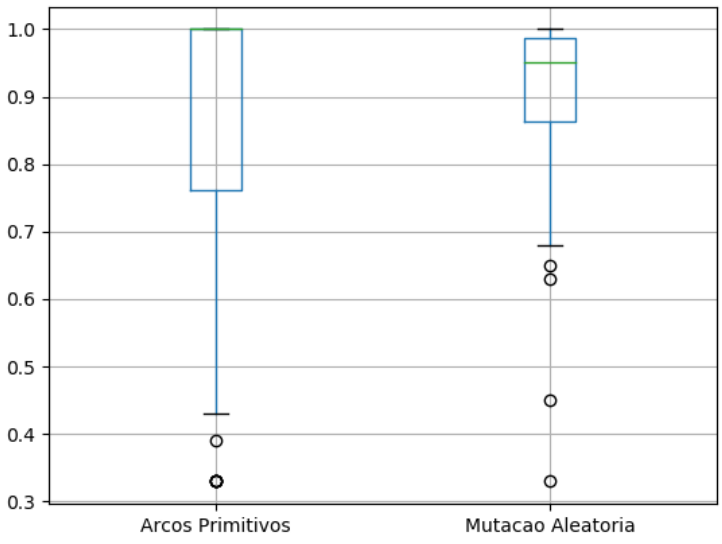


Figura 6.1: Boxplot dos escores de mutação da mutação nos arcos primitivos e da mutação aleatória.

Tabela 6.1: Comparação entre o escore de mutação: mutação nos arcos primitivos x mutação aleatória.

ID	*Função*	$MS_{MIN}(S \cup D)$	$MS_{Random}(S \cup D)$
1	cal	0,57	0,92
2	abss	1	1
3	DefineAndRoundFraction	1	1
4	FnDivide	1	0,91
5	FnNegate	1	1
6	FnTimes	1	0,95
7	checkIt	1	1
8	isPalindrome	0,83	0,98
9	countPositive	0,8	0,82
10	incrdate	0,64	0,77
11	printdate	1	0,93
12	digit_reverser	1	0,97
13	findLast	0,6	0,91
14	findVal	0,8	0,82
15	isSorted	1	0,9
16	less01	1	1
17	less02	1	1
18	show	1	0,83
19	sink	0,75	0,8
20	sort	0,75	0,97
21	invert	1	0,95
22	jday	0,5	0,45
23	jdate	1	0,98
24	lastZero	0,75	0,72
25	lcp	1	1
26	LRS	1	0,88
27	sort	0,33	0,86
28	arraycopy	1	0,98
29	merge	0,71	0,95
30	mergeSort	1	1
31	numZero	1	0,86
32	oddOrPos	0,8	0,87
33	byebye	1	1
34	dayofweek	0,8	0,82
35	dispatch	1	0,98
36	genmonth	0,33	0,95
37	genweek	0,83	0,96
38	getmmddyy	0,33	0,33
39	jan1	0,75	0,65
40	power	1	0,91
41	isDivisible	1	1
42	printPrimes	0,33	0,86
43	quicksort	1	0,94
44	sort	1	0,87
45	addstr	1	1
46	amatch	0,63	0,83
47	change	1	1
48	dodash	1	0,98
49	esc	1	1
50	getccl	1	0,95
51	locate	1	0,8
52	makepat	0,91	0,89
53	makesub	1	1
54	omatch	0,96	0,96
55	patsize	1	0,99
56	putsub	1	1
57	stclose	0,95	0,91
58	subline	1	0,92
59	sum	1	0,97
60	alt_sep_test	1	1
61	Non_Crossing_Biased_Climb	1	0,97
62	Non_Crossing_Biased_Descend	1	0,97
63	pat	0,63	0,9
64	takeOut	1	0,93
65	trash	1	0,98
66	twoPred	0,88	0,87
67	cal	0,33	0,63
68	dispatch	0,39	0,68
69	jan1	0,43	0,72
70	pstr	1	1
	Média	0,86	0,9

Como pode ser observado, os escores de mutação da mutação aleatória possui uma menor variabilidade do que a mutação nos arcos primitivos. Por outro lado, o valor da mediana dos escore de mutação ao utilizar a abordagem dos arcos primitivos é superior à da mutação aleatória (1 > 0.95). Ou seja, a maior parte das funções atingiram um escore de mutação igual à 1 e, consequentemente, foram capazes de matar um maior número de mutantes minimais.

Tabela 6.2: Informação detalhadas do boxplot apresentado.

	Arcos Primitivos	**Mutação Aleatória**
Mínimo	0,33	0,33
1º Quartil	0,76	0,86
Mediana	1	0,95
Média	0,86	0,90
3º Quartil	1	0,98
Máximo	1	1

Ao analisar os resultados de cada função, observa-se que em 31 funções tem-se $MS_{MIN}(S \cup D) > MS_{Random}(S \cup D)$. Em 25 das 31 funções, o escore de mutação utilizando a abordagem dos arcos primitivos foi igual à 1. As mesmas 25 funções tiveram um escore médio de 0,93 ao utilizar a mutação aleatória. Em alguns casos, nota-se que a diferença dos escores entre as abordagens é considerável. Por exemplo, nas funções *show* e *locate* (Linhas 18 e 51, respectivamente), o escore de mutação reduziu de 1 ao considerar a abordagem dos arcos primitivos para 0,83 e 0,80, respectivamente, ao aplicar a mutação aleatória.

Em 19 funções, os escores de ambas abordagens foram os mesmos. Ressalta-se que em 9 delas, não houve redução no número de mutantes, ou seja, o número de mutantes nos arcos primitivos corresponde ao número total de mutantes gerados para a função. Desta forma, independente da estratégia utilizada, o escore de mutação seria 1.

Já em 20 funções, têm-se que $MS_{Random}(S \cup D) > MS_{MIN}(S \cup D)$. Ao considerar apenas as 20 funções, a média do escore mutação foi de 0,85 na mutação aleatória. De forma análoga, a média de escore de mutação ao utilizar a abordagem dos arcos primitivos foi bastante inferior a mutação aleatória: 0,61. Porém, é importante ressaltar que nenhuma dessas funções atingira escore de 1 na mutação aleatória. Por outro lado, nenhuma função atingiu um escore superior a 0,85 na mutação nos arcos primitivos.

Por fim, em 42 funções foi possível atingir o escore de mutação igual à 1 ao utilizar a abordagem da mutação nos arcos primitivos. Em contrapartida, apenas 17 funções atingiram o escore de mutação igual à 1 ao considerar a mutação aleatória. Porém, essas 17 funções também atingiram o escore de mutação de 1 na mutação nos arcos primitivos.

6.4 Considerações Finais

O presente capítulo apresentou a comparação da estratégia proposta nesta dissertação com a mutação aleatória, para analisar se existe ganhos reais ao utilizar o conceito de arcos primitivos no contexto do teste de mutação.

De forma geral, é possível verificar que existe uma leve vantagem ao utilizar a mutação nos arcos primitivos em relação à mutação aleatória. Como visto anteriormente, a mutação nos arcos primitivos obteve um número maior de funções com escore superior à mutação aleatória. Embora o escore médio final da mutação aleatória tenha sido um pouco superior, é importante ressaltar que a mutação nos arcos primitivos concentra um maior número de funções com escore igual à 1, mostrando uma grande vantagem ao matar os mutantes minimais.

Porém, é importante ressaltar que a análise da comparação entre as abordagens considerou apenas o escore de mutação obtido. Entretanto, para apresentar uma análise mais completa sobre os ganhos do uso da abordagem dos arcos primitivos em relação a mutação é necessário avaliar os custos para calcular o conjunto de arcos primitivos. Este custo pode ser avaliado na geração do grafo fluxo de controle, que no caso do experimento apresentado não influencia no custo, visto que a ferramenta Proteum/IM (Delamaro et al., 2000) gera o grafo fluxo de controle no momento da execução do teste de mutação. E, além disso, o custo também pode ser analisado de acordo com a complexidade do algoritmo e do tempo de execução do mesmo.

Por exemplo, ao considerar o grafo fluxo de controle de uma das funções utilizados (*makepat*), verificou-se que o tempo gasto, em segundos, para computar o conjunto de arcos primitivos foi de, aproximadamente, 0,002 segundos[1]. É importante ressaltar que este grafo apresentava, exatamente, 30 nós e 39 arcos.

No entanto, salienta-se que o presente trabalho não realizou a análise detalhada da complexidade do algoritmo para gerar o conjunto de arcos primitivos.

[1] A especificação da máquina utilizada para realização do experimento: processador Intel Core i7 3rd Generation, 8GB de memória RAM, placa de vídeo NVIDIA GT630M, sistema operacional Linux Mint 19 Tara.

Capítulo 7

DEFINIÇÃO PRELIMINIAR DE NOVOS CRITÉRIOS DE TESTE: TESTE DE MUTAÇÃO E ARCOS PRIMITIVOS

7.1 Considerações Iniciais

Até o momento, o trabalho apresentou uma abordagem que combina teste de mutação e informação sobre o fluxo de controle do programa visando reduzir o custo do teste de mutação. De forma geral, os resultados mostraram que foi possível reduzir o número de mutantes e ainda preservar um alto escore de mutação. Além disso, os dados mostraram uma alta concentração de mutantes minimais nos arcos primitivos. Ademais, ao comparar a estratégia proposta no trabalho com a mutação aleatória observa-se que, de forma geral, a mutação nos arcos primitivos apresentou resultados superiores em relação à mutação aleatória.

Agora, o objetivo é apresentar uma definição preliminar de novos critérios de teste a partir dos resultados obtidos até o momento. Como observado na Seção 5.2 e apresentado na Tabela 5.3, existem casos em que o escore de mutação foi superior ao considerar apenas os mutantes do conjunto S, enquanto que em alguns casos o escore de mutação foi superior considerando apenas os mutantes do conjunto D. Desta forma, é possível considerar cada um dos três conjuntos S, D e $S \cup D$ como possíveis critérios de teste.

A Seção 7.2 apresenta uma proposta preliminar de novos critérios, definindo uma análise de inclusão das mesmas. Em seguida, é apresentada uma possível maneira de aplicação dos critérios.

7.2 Estudo e Definição Preliminar de Critérios de Teste

Como descrito anteriormente, os critérios de teste provêm subsídios que guiam o testador a realizar a atividade de maneira efetiva.

Do ponto de vista teórico, os critérios podem ser comparados considerando três perspectivas (Weyuker et al., 1991; Wong et al., 1995):

- **Custo**: Esforço necessário para satisfazer um critério, ou seja, a quantidade de casos de teste necessário para empregar um critério.
- **Eficácia**: Capacidade de um critério em detectar defeitos
- **Dificuldade de Satisfação (ou *strength*)**: Refere-se à probabilidade de satisfazer um critério tendo satisfeito outro critério (Mathur e Eric Wong, 1994).

Para avaliar os critérios de teste e definir uma relação de hierarquia entre eles, é utilizado o conceito de análise de inclusão (Rapps e Weyuker, 1982; Vincenzi, 2004). Sejam C_1 e C_2 dois critérios de teste, então:

- C_1 *inclui* C_2, se todo conjunto de teste $C_1 - adequado$ for $C_2 - adequado$.
- C_1 *inclui estritamente* C_2 ($C1 \rightarrow C_2$), se C_1 inclui C_2 e C_2 não inclui C_1.
- C_1 e C_2 são *incomparáveis* se não existir relação de inclusão entre eles.

A seguir, são apresentados os critérios definidos a partir dos dados resultados obtidos.

7.2.1 Definição Preliminar de Novos Critérios de Teste

Até o momento, os dados mostraram que, no geral, foi possível alcançar um alto escore de mutação ao considerar os mutantes localizados nos arcos primitivos. Porém, ao analisar cada função de forma individual, é possível observar que em alguns casos o escore de mutação é superior ao considerar apenas os mutantes do conjunto S. Em outros casos, o escore de mutação é superior ao considerar apenas os mutantes de D.

A partir desta observação, definem-se três critérios:

- **mutação-nó-origem-arco-primitivo**: mutação nos nós de origem dos arcos primitivos, ou seja, os mutantes são gerados apenas nos nós de origem dos arcos primitivos.

- **mutação-nó-destino-arco-primitivo**: mutação nos nós de destino dos arcos primitivos, ou seja, os mutantes são gerados apenas nos nós de destino dos arcos primitivos.
- **mutação-arco-primitivo**: mutação nos arcos primitivos, ou seja, os mutantes são gerados nos nós de origem e nos nós de destino dos arcos primitivos.

7.2.2 Análise de Inclusão dos Critérios Propostos

Como descrito anteriormente, a análise de inclusão é utilizada para avaliar critérios e definir uma hierarquia entre os mesmos.

No Capítulo 5.2 a Tabela 5.3 mostrou que existem casos em $MS_{PA}(S) > MS_{PA}(D)$ e casos em que $MS_{PA}(D) > MS_{PA}(S)$. Em 34 das 70 funções, o escore mutação foi superior ao utilizar apenas os mutantes localizados nos nós de origem, enquanto que em 23 das 70 funções o escore de mutação foi superior ao considerar apenas os mutantes dos nós de destino. Além disso, em 13 funções os escores obtidos foram iguais.

Desta forma, não é possível definir uma relação de inclusão entre os critérios *mutação-nó-origem-arco-primitivo* e *mutação-nó-destino-arco-primitivo* e, portanto, conclui-se que ambos os critérios são **incomparáveis**.

Por outro lado, o conjunto de mutantes nos arcos primitivos, ou seja, do conjunto $S \cup D$ é composto pelos mutantes localizados nos nós de origem e nos nós de destino. Por conseguinte, $MS_{PA}(S \cup D) \geq MS_{PA}(S)$ e $MS_{PA}(S \cup D) \geq MS_{PA}(D)$. Assim, é possível concluir que o critério *mutação-arco-primitivo* **inclui** os outros dois critérios definidos anteriormente.

A Figura 7.1 ilustra as relações de inclusão entre os critérios propostos.

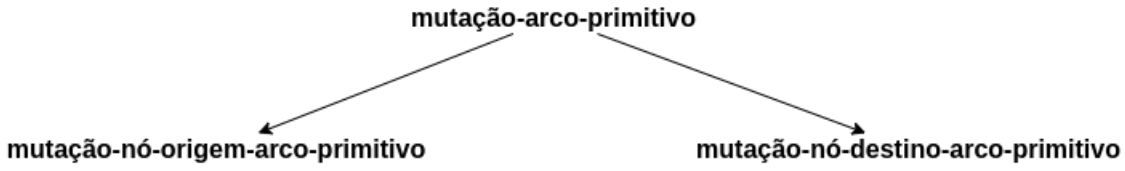


Figura 7.1: Relação de inclusão dos critérios de teste propostos.

7.2.3 Definição Preliminar de uma Estratégia para Aplicação dos Critérios Propostos

A estratégia de aplicação preliminar dos critérios foi definida de acordo com a relação de inclusão apresentado na Seção 7.2.2. Para os critérios incomparáveis foi computado o

strength de ambos os critérios, visando definir uma possível ordem de aplicação. Para computar o *strength* foram realizados os seguintes passos:

- Selecionar os casos de teste T_S e T_D que matam os mutantes dos conjuntos S e D, respectivamente.
- Executar os casos de teste T_S e T_D nos conjuntos de mutantes D e S, respectivamente, e computar o escore de mutação.

A Tabela 7.1 apresenta os resultados obtidos. A coluna $T_D \rightarrow S$ mostra o escore de mutação ao aplicar os casos de teste T_D no conjunto S e a coluna $T_S \rightarrow D$ apresenta o escore de mutação ao aplicar T_S em D.

Os dados mostram que o escore médio final do conjunto de teste T_S em relação ao escore médio do conjunto T_D (0,94 > 0,91). Desta forma, o conjunto T_S possui um *strength* superior ao conjunto T_D.

Assim, uma possível estratégia de aplicação seria: após a geração dos arcos primitivos para um dado programa em teste, gerar apenas mutantes nos nós de origem dos arcos primitivos e desenvolver requisitos de teste para matar esses mutantes. Caso o testador julgar necessário, gerar mutantes nos nós de destino dos arcos primitivos e desenvolver requisitos de teste para matar esses novos mutantes.

Vale ressaltar, que essa estratégia preliminar se baseia apenas na relação de inclusão e no escore de mutação após o cálculo do *strength*. Entretanto, algumas características estruturais do programa, como por exemplo, expressões com múltiplas condições ou múltiplos operadores lógicos, podem influenciar nos dados resultantes.

Posto isso, uma melhoria para a estratégia é verificar se é possível relacionar os critérios propostos com métricas de análise estática do software e, a partir disso, extrair informações sobre as características estruturais do software e avaliar qual critério melhor se adapta para os programas com tais características.

Porém, para validar essa questão é necessário realizar um estudo experimental mais amplo e uma análise mais profunda nos dados.

7.3 Considerações Finais

O presente capítulo apresentou uma definição preliminar de possíveis novos critérios de teste que envolvam teste de mutação e arcos primitivos, além de uma estratégia preliminar para aplicação dos critérios propostos.

A definição dos critérios baseou-se exclusivamente nos resultados relacionados ao escore de mutação obtido no estudo experimental. A partir dos resultados, foi possível propor três novos critérios e as suas relações de inclusão.

Tabela 7.1: Escore de mutação considerando o *strength* dos conjuntos S e D.

ID	*Função*	$T_D \rightarrow S$	$T_S \rightarrow D$
1	cal	1	0,99
2	abss	0,97	1
3	DefineAndRoundFraction	0,95	1
4	FnDivide	0,9	1
5	FnNegate	1	1
6	FnTimes	0,99	0,97
7	checkIt	0,52	1
8	isPalindrome	0,96	1
9	countPositive	0,44	1
10	incrdate	0,91	0,91
11	printdate	0,88	0,41
12	digit_reverser	1	0,81
13	findLast	0,9	1
14	findVal	0,95	1
15	isSorted	0,73	1
16	less01	0,98	1
17	less02	1	1
18	show	0,97	0,83
19	sink	1	0,96
20	sort	0,92	0,96
21	invert	0,88	0,96
22	jday	0,96	1
23	jdate	0,86	0,96
24	lastZero	0,87	1
25	lcp	1	0,62
26	LRS	0,98	0,98
27	sort	0,94	1
28	arraycopy	0,89	0,94
29	merge	0,93	1
30	mergeSort	0,86	0,89
31	numZero	0,67	1
32	oddOrPos	0,3	1
33	byebye	1	0,9
34	dayofweek	1	0,95
35	dispatch	0,69	0,78
36	genmonth	0,98	0,99
37	genweek	1	1
38	getmmddyy	0,87	0,91
39	jan1	0,74	1
40	power	0,99	0,94
41	isDivisible	1	1
42	printPrimes	0,97	1
43	quicksort	1	0,98
44	sort	0,93	1
45	addstr	1	1
46	amatch	0,99	0,97
47	change	1	1
48	dodash	1	0,95
49	esc	0,98	1
50	getccl	1	0,83
51	locate	1	0,98
52	makepat	0,99	0,93
53	makesub	1	0,79
54	omatch	0,88	1
55	patsize	1	0,45
56	putsub	1	1
57	stclose	1	0,96
58	subline	1	1
59	sum	0,97	1
60	alt_sep_test	0,55	1
61	Non_Crossing_Biased_Climb	0,98	0,98
62	Non_Crossing_Biased_Descend	0,99	0,65
63	pat	0,96	1
64	takeOut	0,97	0,98
65	trash	0,87	1
66	twoPred	0,71	1
67	cal	1	0,99
68	dispatch	0,72	1
69	jan1	1	0,95
70	pstr	1	1
	Média	0,91	0,94

Por fim, a estratégia de aplicação dos critérios foi definida a partir do *strength* do conjunto de casos de teste dos conjuntos S e D, como forma apresentar uma possível ordem de aplicação dos critérios. Como ressaltado, existem diversas melhorias que podem ser realizadas, como por exemplo, definir uma ordem de aplicação dos critérios com base em métricas de análise estática do software.

Capítulo 8

CONCLUSÃO

8.1 Conclusão, Contribuições e Limitações

Testar um software não é uma tarefa trivial. Por conta disso, existem diversas técnicas e critérios que fornecem subsídios ao testador, para que o teste seja realizado de forma efetiva, minimizando os potenciais defeitos, provendo um produto de maior qualidade.

Este trabalho abordou o teste de mutação (DeMillo et al., 1978; Hamlet, 1977) que é um critério do teste baseado em defeitos, utilizado para avaliar a qualidade de um conjunto de teste. Porém, o teste de mutação possui duas grandes desvantagens: o alto custo computacional para gerar e executar os mutantes e a existência de mutantes equivalentes. Neste contexto, diversos pesquisas vem sendo realizadas, buscando reduzir o custo do teste de mutação e torná-lo aplicável em situações práticas (Pizzoleto et al., 2019).

Este trabalho apresenta uma abordagem que combina teste de mutação e informação de fluxo de controle visando reduzir o custo do teste de mutação. De forma geral, o objetivo é verificar se ao executar apenas mutantes localizados em partes específicas do código, dado por um conjunto de arcos primitivos do programa em teste, é possível reduzir o número de mutantes executados e ainda preservar um alto escore de mutação.

Em relação aos resultados, os resultados mostraram que, no geral, foi possível reduzir o número de mutantes executados e atingir um alto escore de mutação. Por exemplo, considerando apenas o conjunto S, houve uma redução do número de mutantes executados por função de, aproximadamente, 60%. Neste caso, o escore médio final do conjunto S foi de 0,95. Já no conjunto D, houve uma redução média de 71,86% no número de mutantes executados por função e um escore médio final de 0,93. Por fim, ao considerar todos os mutantes nos arcos primitivos, ou seja, os mutantes do conjunto $S \cup D$, houve uma redução de aproximadamente 32% de mutantes por função e um escore médio final de 0,98.

Além disso, os dados mostram que a abordagem proposta também reduziu o número de mutantes equivalentes. Ao considerar os mutantes nos arcos primitivos ($S \cup D$), houve uma redução de 34% do número de mutantes equivalentes.

Doravante, uma segunda contribuição está relacionada ao trabalho de Delamaro et al. (2018), no qual os autores sugerem que a geração dos mutantes minimais pode estar ligado à estrutura do programa, uma vez que não é possível relacionar a geração destes mutantes a um conjunto de operadores de mutação. Os resultados mostraram que, no geral, houve uma alta concentração de mutantes minimais nos arcos primitivos, ou seja, 70% dos mutantes minimais estavam localizados em um nó de origem ou em um nó de destino.

A seguir, a mutação nos arcos primitivos foi comparada com uma abordagem já existente: mutação aleatória. Os resultados mostraram que a mutação nos arcos primitivos obteve, em termos de escore mutação, uma pequena vantagem em relação à mutação aleatória.

Por fim, o trabalho apresentou três novos critérios de teste a partir dos resultados obtidos e definiu a relação de inclusão entre eles:

- mutação-nó-origem-arco-primitivo.
- mutação-nó-destino-arco-primitivo.
- mutação-arco-primitivo.

Ademais, o trabalho mostrou uma possível forma de aplicação preliminar dos critérios propostos.

Em relação as limitações, um ponto importante a ser ressaltado é que o estudo experimental realizado utilizou apenas um conjunto de teste adequado para cada um dos programas. Por este motivo, a utilização de diferentes conjuntos pode apresentar resultados distintos aos obtidos.

Outra limitação está relacionada aos programas utilizados. Embora o conjunto de 70 funções seja representativo, uma vez que o teste de mutação foi aplicado a nível de unidade, a utilização de programas mais complexos e com um número maior de unidades, pode apresentar diferentes resultados.

8.2 Trabalhos Futuros

Em relação aos trabalhos futuros, pode ser destacado:

- Realizar o experimento com diferentes conjuntos de teste para verificar se os resultados se mantém.

- Realizar o experimento com programas mais complexos.
- Aplicar o mesmo experimento utilizando o algoritmo *REHFLUXDA* (Chaim, 1991).
- Estender o experimento utilizando outras linguagens de programação, como por exemplo, Java ou Python.
- Conforme apresentado na Seção 7.2.3, algumas características estruturais do programa podem influenciar nos dados resultantes. Desta forma, a utilização de métricas de análise estática pode apresentar características de cada programa e, partir disto, será possível definir estratégias para aplicação das técnicas de redução do custo do teste de mutação de acordo com essas características.

8.3 Publicação de Artigos

Além das contribuições apresentadas anteriormente, este trabalho gerou um artigo científico publicado e apresentado, conforme detalhado abaixo:

- Artigo: **Reducing the Cost of Mutation Testing with the Use of Primitive Arcs Concept**
- Autores: Pedro Henrique Kuroishi, Márcio Eduardo Delamaro, José Carlos Maldonado e Auri Marcelo Rizzo Vincenzi
- Evento: XIX Simpósio Brasileiro de Qualidade de Software (SQBS)
- Ano: 2020
- DOI: https://doi.org/10.1145/3439961.3439981

REFERÊNCIAS

ACREE, A. T.; BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. *Mutation analysis*. Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta/GA - USA, 1979.

ACREE, JR., A. T. *On mutation*. PhD Thesis, Atlanta, GA, USA, aAI8107280, 1980.

AGRAWAL, H. Dominators, super blocks, and program coverage. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, New York, NY, USA: Association for Computing Machinery, p. 25–34, 1994 (*POPL '94*, v.).

AMMANN, P.; DELAMARO, M. E.; OFFUTT, J. Establishing theoretical minimal sets of mutants. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, Washington, DC, USA: IEEE Computer Society, p. 21–30, 2014 (*ICST '14*, v.).

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R. Towards the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, v. 11, n. 2, p. 113–136, 2001.

BOEHM, B.; BASILI, V. R. Software defect reduction top 10 list. *Computer*, v. 34, n. 1, p. 135–137, 2001.

BUDD, T. A.; ANGLUIN, D. Two notions of correctness and their relation to testing. *Acta Inf.*, v. 18, n. 1, p. 31–45, 1982.

CHAIM, M. L. *Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. MSc Dissertation, DCA/FEEC/UNICAMP, Campinas, SP, 1991.

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.

COPELAND, L. *A practitioner's guide to software test design*. Norwood, MA, USA: Artech House, Inc., 2003.

DELAMARO, M.; CHAIM, M. L.; MALDONADO, J. C. Where are the minimal mutants? In: *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, New York, NY, USA: ACM, p. 190–195, 2018 (*SBES '18*, v.).

DELAMARO, M. E.; DENG, L.; DURELLI, V. H. S.; LI, N.; OFFUTT, J. Experimental evaluation of sdl and one-op mutation for c. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, Washington, DC, USA: IEEE Computer Society, p. 203–212, 2014 (*ICST '14*, v.).

DELAMARO, M. E.; DENG, L.; LI, N.; DURELLI, V.; OFFUTT, J. Growing a reduced set of mutation operators. In: *2014 Brazilian Symposium on Software Engineering*, p. 81–90, 2014.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. Introdução ao teste de software. 2016.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Proteum/IM 2.0: An integrated mutation testing environment. In: *Mutation 2000 Symposium*, San Jose, CA: Kluwer Academic Publishers, p. 91–101, 2000.

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing deletion mutation operators. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, Washington, DC, USA: IEEE Computer Society, p. 11–20, 2014 (*ICST '14*, v.).

DELGADO-PÉREZ, P.; HABLI, I.; GREGORY, S.; ALEXANDER, R.; CLARK, J.; MEDINA-BULO, I. Evaluation of mutation testing in a nuclear industry case study. *IEEE Transactions on Reliability*, v. 67, n. 4, p. 1406–1419, 2018.

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.

DEMILLO, R. A.; MCCRACKEN, W. M.; MARTIN, R. J.; PASSAFIUME, J. F. *Software testing and evaluation*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1987.

DURELLI, V. H.; DELAMARO, M. E.; OFFUTT, J. An experimental comparison of edge, edge-pair, and prime path criteria. *Science of Computer Programming*, v. 152, p. 99 – 115, 2018.

GHEYI, R.; RIBEIRO, M.; SOUSA, B.; GUIMARÃES, M.; FERNANDES, L.; D'AMORIM, M.; ALVES, V.; TEIXEIRA, L.; FONSECA, B. Identifying method-level mutation subsumption relations using z3. *Information and Software Technology*, v. 132, p. 106496, 2021.

GOPINATH, R.; AHMED, I.; ALIPOUR, M. A.; JENSEN, C.; GROCE, A. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, v. 66, n. 3, p. 854–874, 2017.

HAMLET, R. G. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, v. 3, n. 4, p. 279–290, 1977.

HARMAN, M.; HIERONS, R.; DANICIC, S. *The relationship between program dependence and mutation analysis* USA: Kluwer Academic Publishers, p. 5–13, 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, p. 61–72, 2000 (*ICSE '00*, v.).

HOWDEN, W. E. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, v. SE-4, n. 4, p. 293–298, 1978.

IEEE Iso/iec/ieee international standard - systems and software engineering–vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, p. 1–541, 2017.

JI, C.; CHEN, Z.; XU, B.; ZHAO, Z. A novel method of mutation clustering based on domain analysis. In: *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering, SEKE 2009*, p. 422–425, 2009.

JUST, R.; KURTZ, B.; AMMANN, P. Inferring mutant utility from program context. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, New York, NY, USA: ACM, p. 284–294, 2017 (*ISSTA 2017*, v.).

KINTIS, M.; PAPADAKIS, M.; MALEVRIS, N. Employing second-order mutation for isolating first-order equivalent mutants. *Softw. Test. Verif. Reliab.*, v. 25, n. 5–7, p. 508–535, 2015.

KURTZ, B.; AMMANN, P.; DELAMARO, M. E.; OFFUTT, J.; DENG, L. Mutant subsumption graphs. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, p. 176–185, 2014.

KURTZ, B.; AMMANN, P.; OFFUTT, J. Static analysis of mutant subsumption. In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 1–10, 2015.

KURTZ, B.; AMMANN, P.; OFFUTT, J.; DELAMARO, M. E.; KURTZ, M.; GÖKÇE, N. Analyzing the validity of selective mutation with dominator mutants. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA: Association for Computing Machinery, p. 571–582, 2016 (*FSE 2016*, v.).

M. HERMAN, P. A data flow analysis approach to program testing. *Australian Computer Journal*, v. 8, p. 92–96, 1976.

MALDONADO, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. PhD Thesis, DCA/FEE/UNICAMP, Campinas, SP, 1991.

MARCOZZI, M.; BARDIN, S.; KOSMATOV, N.; PAPADAKIS, M.; PREVOSTO, V.; CORRENSON, L. Time to clean your test objectives. In: *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA: ACM, p. 456–467, 2018 (*ICSE '18*, v.).

MATHUR, A.; ERIC WONG, W. An empirical comparison of mutation and data flow based test adequacy criteria. *Software Testing, Verification and Reliability*, v. 4, 1994.

MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing. In: *[1991] Proceedings The Fifteenth Annual International Computer Software Applications Conference*, p. 604–605, 1991.

MCCABE, T. J. A complexity measure. *IEEE Trans. Softw. Eng.*, v. 2, n. 4, p. 308–320, 1976.

MCMINN, P.; WRIGHT, C. J.; MCCURDY, C. J.; KAPFHAMMER, G. M. Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas. *IEEE Transactions on Software Engineering*, v. 45, n. 5, p. 427–463, 2019.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. 3rd ed. Wiley Publishing, 2011.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, v. 5, n. 2, p. 99–118, 1996.

OFFUTT, A. J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. *Softw. Test., Verif. Reliab.*, v. 7, p. 165–192, 1997.

OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: *Proceedings of 1993 15th International Conference on Software Engineering*, p. 100–107, 1993.

OFFUTT, A. J.; UNTCH, R. H. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, Norwell, MA, USA: Kluwer Academic Publishers, p. 34–44, 2001.

PAPADAKIS, M.; DELAMARO, M.; LE TRAON, Y. Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*, v. 95, p. 298 – 319, special Section: ACM SAC-SVT 2013 + Bytecode 2013, 2014.

PAPADAKIS, M.; LE TRAON, Y. Mutation testing strategies using mutant classification. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, New York, NY, USA: Association for Computing Machinery, p. 1223–1229, 2013 (*SAC '13*, v.).

PATEL, K.; HIERONS, R. M. Resolving the equivalent mutant problem in the presence of non-determinism and coincidental correctness. In: WOTAWA, F.; NICA, M.; KUSHIK, N., eds. *Testing Software and Systems*, Cham: Springer International Publishing, p. 123–138, 2016.

PATRICK, M.; ORIOL, M.; CLARK, J. A. Messi: Mutant evaluation by static semantic interpretation. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, p. 711–719, 2012.

PETROVIC, G.; IVANKOVIC, M. State of mutation testing at google. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, p. 163–171, 2017.

PETROVIC, G.; IVANKOVIC, M.; KURTZ, B.; AMMANN, P.; JUST, R. An industrial application of mutation testing: Lessons, challenges, and research directions. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, p. 47–53, 2018.

PIZZOLETO, A. V.; FERRARI, F. C.; OFFUTT, J.; FERNANDES, L.; RIBEIRO, M. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, v. 157, p. 110388, 2019.

PRESSMAN, R. *Software engineering: A practitioner's approach*. 7 ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, Los Alamitos, CA, USA: IEEE Computer Society Press, p. 272–278, 1982 (*ICSE '82*, v.).

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, v. 11, n. 4, p. 367–375, 1985.

REID, S. C. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: *Proceedings Fourth International Software Metrics Symposium*, p. 64–73, 1997.

SCHULER, D.; DALLMEIER, V.; ZELLER, A. Efficient mutation testing by checking invariant violations. In: *Proceedings of the Eighteenth International Symposium*

on Software Testing and Analysis, ISSTA '09, New York, NY, USA: Association for Computing Machinery, p. 69–80, 2009 (*ISSTA '09*, v.).

SCHULER, D.; ZELLER, A. (un-)covering equivalent mutants. In: *2010 Third International Conference on Software Testing, Verification and Validation*, p. 45–54, 2010.

SUN, C.-A.; XUE, F.; LIU, H.; ZHANG, X. A path-aware approach to mutant reduction in mutation testing. *Inf. Softw. Technol.*, v. 81, n. C, p. 65–81, 2017.

VINCENZI, A. M. R. *Orientação a objeto: Definição, implementação e análise de recursos de teste e validação*. Tese de doutoramento, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, 2004.

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 the Brazilian Computer Society*, v. 12, n. 2, p. 49–61, 2006.

WEYUKER, E. J.; WEISS, S. N.; HAMLET, D. Comparison of program testing strategies. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, New York, NY, USA: Association for Computing Machinery, p. 1–10, 1991 (*TAV4*, v.).

WONG, W. E.; MATHUR, A. P. Reducing the cost of mutation testing: An empirical study. *J. Syst. Softw.*, v. 31, n. 3, p. 185–196, 1995.

WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. *Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness* Boston, MA: Springer US, p. 258–265, 1995.