JADER ELIAS MEROS

PRIORIZAÇÃO DE TESTES DE SISTEMA AUTOMATIZADOS POR MEIO DE GRAFOS DE CHAMADAS

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Curitiba PR
Fevereiro de 2016

JADER ELIAS MEROS

PRIORIZAÇÃO DE TESTES DE SISTEMA AUTOMATIZADOS POR MEIO DE GRAFOS DE CHAMADAS

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de Mestre em Computação Aplicada.

Área de concentração: *Engenharia de Sistemas Computacionais*

Orientador: Maria Cláudia Figueiredo Pereira Emer
Co-orientador: Adolfo Gustavo Serra Seca Neto

Curitiba PR
Fevereiro de 2016

Dados Internacionais de Catalogação na Publicação

M567p Meros, Jader Elias
2016 Priorização de testes de sistema automatizados por meio de grafos de chamadas / Jader Elias Meros.-- 2016.
79 p.: il.; 30 cm

Texto em português, com resumo em inglês.
Dissertação (Mestrado) - Universidade Tecnológica Federal do Paraná. Programa de Pós-Graduação em Computação Aplicada, Curitiba, 2016.
Bibliografia: p. 53-59.

1. Software - Testes - Automação - Estudo de casos. 2. Desenvolvimento ágil de software. 3. Teoria dos grafos. 4. Falhas de sistemas de computação - Detecção. 5. Métodos de simulação. 6. Engenharia de software. 7. Computação - Dissertações. I. Emer, Maria Cláudia Figueiredo Pereira, orient. II. Seca Neto, Adolfo Gustavo Serra, coorient. III. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Computação Aplicada. IV. Título.

CDD: Ed. 22 -- 621.39

Biblioteca Central da UTFPR, Câmpus Curitiba

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO Nº 41

Aos 31 dias do mês de março de 2016 realizou-se na sala C 301 a sessão pública de Defesa da Dissertação de Mestrado intitulada "Priorização de Testes de Sistema Automatizados por Meio de Grafos de Chamadas", apresentada pelo aluno **Jader Elias Meros** como requisito parcial para a obtenção do título de Mestre em Computação Aplicada, na área de concentração "Engenharia de Sistemas Computacionais", linha de pesquisa "Engenharia de Software".

Constituição da Banca Examinadora:

Profª Drª. Maria Claudia F. Pereira Emer, UTFPR - CT (Presidente) ___________________

Profª. Drª. Thelma Elita Colanzi Lopes- (UEM) ____________________________________

Prof. Dr. Laudelino Cordeiro Bastos, UTFPR- CT __________________________________

Em conformidade com os regulamentos do Programa de Pós-Graduação em Computação aplicada e da Universidade Tecnológica Federal do Paraná, o trabalho apresentado foi considerado ___________________________ (aprovado/reprovado) pela banca examinadora. No caso de aprovação, a mesma está condicionada ao cumprimento integral das exigências da banca examinadora, registradas no verso desta ata, da entrega da versão final da dissertação em conformidade com as normas da UTFPR e da entrega da documentação necessária à elaboração do diploma, em até ____________ dias desta data.

Ciente (assinatura do aluno): ____

(para uso da coordenação)

A Coordenação do PPGCA/UTFPR declara que foram cumpridos todos os requisitos exigidos pelo programa para a obtenção do título de Mestre.

Curitiba PR, _____/_____/___________ _______________________________

"A Ata de Defesa original está arquivada na Secretaria do PPGCA".

Dedico este trabalho a minha esposa que esteve sempre ao meu lado e ao meu filho que me deu a força e a alegria necessárias para alcançar este objetivo.

Agradecimentos

Antes de nominar os principais personagens que fizeram parte dos últimos 2 anos e meio e que colaboraram diretamente para que este objetivo fosse alcançado, gostaria de agradecer de maneira mais geral a todos os meus amigos e familiares. Sem eles, nada disso seria possível.

Eu não poderia deixar de agradecer a minha orientadora, a Dra. Maria Cláudia Figueiredo Pereira Emer. Não tenho dúvida que a sua dedicação e o seu conhecimento tiveram valor inestimável a este trabalho.

Também o meu agradecimento ao meu co-orientador, o Dr. Adolfo Gustavo Serra Seca Neto. Suas revisões e apontamentos foram certamente importantíssimos para que este trabalho.

Estendo também meus agradecimentos aos demais professores e corpo técnico do Programa de Pós-Graduação em Computação Aplicada da UTFPR, todos certamente tem sua parcela de importância para que este programa seja executado com a qualidade que a universidade pública brasileira merece.

Ao colega, Wilson Bissi, meu sincero agradecimento pelo companheirismo e pela inúmeras dúvidas sanadas durante todo este tempo.

Ao Tribunal Regional do Trabalho da 9ª Região, a Unify Communications e aos meus colegas de trabalho, meu agradecimento pela flexibilidade e apoio nos momentos em que precisei me ausentar do trabalho para executar as atividades necessárias para este projeto.

Por fim, o agradecimento a pessoa mais importante da minha vida. A minha esposa que foi quem me incentivou a começar, foi mãe/pai/empregada quando não pude estar presente e ainda assim me apoiou incondicionalmente durante todo este tempo. Meu mais sinceros agradecimentos, este trabalho certamente também é seu e eu não chegaria até aqui sem você.

Resumo

Com a necessidade cada vez maior de agilizar a entrega de novos desenvolvimentos ao cliente e de diminuir o tempo de desenvolvimento das aplicações, a priorização de casos de teste possibilita a detecção das falhas presentes na aplicação mais rapidamente por meio da ordenação dos casos de teste a serem executados. E, com isso, possibilita também que a correção destas falhas inicie o mais brevemente possível. Entretanto, quando os casos de teste a serem priorizados são testes automatizados de sistema, critérios tradicionais utilizados na literatura como cobertura de código ou modelos do sistema deixam de ser interessantes, dada a característica inerente deste tipo de teste na qual a organização e a modelagem adotadas são ignoradas por se tratarem de testes de caixa preta. Considerando a hipótese de que casos de teste automatizados grandes testam mais partes da aplicação e que casos de teste similares podem estar testando a mesma área da aplicação, parece válido crer que a execução dos casos de teste de sistema priorizando os testes mais complexos pode alcançar resultados positivos quando comparada à execução não ordenada dos casos de teste. É neste cenário que este trabalho propõe o uso dos grafos de chamadas dos próprios casos de teste como critério para priorização destes, priorizando assim a execução dos casos de teste com a maior quantidade de nós no seu grafo. A abordagem proposta neste trabalho mostrou, por meio de dois estudos de caso, ser capaz de melhorar a taxa de detecção de falhas em relação à execução não ordenada dos casos de teste. Além disso, a abordagem proposta obteve resultados semelhantes as abordagens tradicionais de priorização utilizando cobertura de código da aplicação.

Palavras-chave: priorização de casos de teste, grafo de chamadas, teste de sistema.

Abstract

With the increasing need to streamline the delivery of new developments to the customer and reduce application development time, test case prioritization allows a quicker detection of faults present in the application through the ordering of test cases to be executed. Besides that, a quicker detection enables also the correction of these faults to start as soon as possible. However, when the test cases to be prioritized are automated system tests, traditional criteria used in the literature like code coverage or system models become uninteresting, given that this type of test case, classified as black box test, ignores how the application was coded or modeled. Considering the hypothesis that bigger automated test cases verify more parts of the application and that similar test cases may be testing the same application areas, it seems valid to believe that giving a higher priority to more complex test cases to be executed first can accomplish positive results when compared to the unordered execution of test cases. It is on this scenario that this project studies the usage of call graphs from test cases as the criterion to prioritize them, increasing the priority of the execution of test cases with the higher number of nodes on the graph. The approach proposed in this document showed through two case studies that it is capable of improving fault detection rate compared to unordered test cases. Furthermore, the proposed approach achieved similar results when compared to a traditional prioritization approach using code coverage of the application.

Keywords: test case prioritization, call graph, system testing.

Sumário

Resumo xi

Abstract xiii

Lista de Figuras xvii

Lista de Tabelas xix

Lista de Abreviações xx

1 Introdução 1
- 1.1 Contexto . . . 1
- 1.2 Motivação . . . 3
- 1.3 Objetivos . . . 3
- 1.4 Contribuições . . . 3
- 1.5 Organização do Texto . . . 4

2 Fundamentação Teórica 5
- 2.1 Priorização de casos de teste . . . 5
  - 2.1.1 Critérios utilizados para a priorização . . . 5
  - 2.1.2 Algoritmos utilizados para a priorização . . . 7
  - 2.1.3 Métricas usadas para avaliação dos resultados . . . 8
  - 2.1.4 Origem das falhas usadas para avaliar as abordagens . . . 11
  - 2.1.5 Aplicações usadas para avaliar as abordagens . . . 11
- 2.2 Teste automatizado . . . 12
- 2.3 Trabalhos correlatos . . . 13
- 2.4 Considerações finais . . . 14

3 Método de Pesquisa 15
- 3.1 Sobre o método de pesquisa utilizado neste trabalho . . . 15
- 3.2 Sobre o método de pesquisa utilizado para avaliar a proposta . . . 16
  - 3.2.1 Primeiro estudo de caso . . . 16
  - 3.2.2 Segundo estudo de caso . . . 18
  - 3.2.3 Diferença entre a abordagem proposta e a execução dos estudos de caso 19
- 3.3 Considerações finais . . . 19

4 Priorização de Casos de Teste de Sistema Automatizados 21
4.1 Abordagem proposta 21
4.1.1 Sobre a priorização neste cenário 21
4.1.2 Sobre o critério utilizado na priorização 22
4.1.3 Sobre os algoritmos utilizados na priorização 22
4.1.4 Sobre a aplicação da priorização 23
4.2 Ferramentas auxiliares 24
4.2.1 Ferramenta para gerar o grafo de chamadas 24
4.2.2 Ferramenta para obter a cobertura de código 25
4.3 Implementação dos algoritmos de priorização 25
4.4 Considerações finais 26

5 Estudos de Caso 29
5.1 Primeiro estudo de caso 29
5.1.1 Busca por um projeto 29
5.1.2 Preparação do projeto 29
5.1.3 Coleta dos dados 35
5.1.4 Aplicação das abordagens escolhidas 35
5.1.5 Avaliação dos resultados obtidos 40
5.2 Segundo estudo de caso 41
5.2.1 Coleta dos dados 42
5.2.2 Aplicação da abordagem proposta 45
5.2.3 Avaliação dos resultados obtidos 46
5.3 Discussão 46
5.4 Considerações finais 49

6 Conclusões 51

A Estudo de mapeamento da área de pesquisa 61
A.1 Método de pesquisa 61
A.1.1 Estratégia de pesquisa 61
A.1.2 Esquema de classificação 62
A.2 Mapeamento 62

Lista de Figuras

2.1 Exemplo de um grafo de chamadas. Fonte: Grove et al. (1997) 6
2.2 APFD para conjunto de teste priorizado T_1. Fonte: Elbaum, Malishevsky e Rothermel (2000) 9
2.3 APFD para conjunto de teste priorizado T_2. Fonte: Elbaum, Malishevsky e Rothermel (2000) 9
2.4 APFD para conjunto de teste priorizado T_3. Fonte: Elbaum, Malishevsky e Rothermel (2000) 10

3.1 Diagrama de atividades para a execução deste trabalho 15
3.2 Diagrama de atividades do primeiro estudo de caso 17
3.3 Diagrama de atividades do segundo estudo de caso 18

4.1 Diagrama de atividades com os passos executados para a priorização 23

5.1 Aplicação testada no primeiro estudo de caso 30
5.2 Grafo gerado para o caso de teste da opção de menu *File->New* 36
5.3 Grafo gerado para o caso de teste que abre um arquivo usando a opção de menu *File->Open* 37
5.4 Grafo gerado para o caso de teste que adiciona um ThreadGroup usando a opção de menu *Edit->Add->Thread Group* 38
5.5 APFD para as ordens de execução avaliadas no primeiro estudo de caso 41
5.6 APFD para as ordens de execução avaliadas no segundo estudo de caso 47

A.1 Distribuição das pesquisas por critério para a priorização 63
A.2 Distribuição dos algoritmos utilizados para a priorização 63
A.3 Distribuição das métricas de avaliação utilizadas na priorização 64
A.4 Distribuição dos objetos de estudo utilizadas na priorização 64

Lista de Tabelas

2.1 Casos de teste e lista de falhas expostas. Fonte: Elbaum, Malishevsky e Rothermel (2000) . . . 8

4.1 Comparativo entre ferramentas para cálculo da cobertura de código . . . 25

5.1 Caso de teste para criar um novo arquivo usando opção do menu *File* . . . 30
5.2 Caso de teste para abrir um arquivo usando opção do menu *File* . . . 31
5.3 Caso de teste para adicionar um *Thread Group* ao *Test Plan*. . . . 31
5.4 Falhas semeadas na versão escolhida para o primeiro estudo de caso . . . 34
5.5 Ordem de execução original dos casos de teste no primeiro estudo de caso . . . 35
5.6 Ordem de execução para a abordagem proposta usando guloso total no primeiro estudo de caso . . . 39
5.7 Ordem de execução para a abordagem proposta usando guloso adicional no primeiro estudo de caso . . . 39
5.8 Ordem de execução para a abordagem tradicional usando guloso total no primeiro estudo de caso . . . 40
5.9 Ordem de execução para a abordagem tradicional usando guloso adicional no primeiro estudo de caso . . . 40
5.10 Falhas detectadas pelos casos de teste no segundo estudo de caso . . . 44
5.11 Ordem de execução original dos casos de teste no segundo estudo de caso . . . 44
5.12 Ordem de execução para a abordagem proposta usando guloso total no segundo estudo de caso . . . 45
5.13 Ordem de execução para a abordagem proposta usando guloso adicional no segundo estudo de caso . . . 46

A.1 Classificação detalhada dos estudos mapeados . . . 66
A.2 Informações sobre os estudos mapeados . . . 70

Lista de Abreviações

APBC	*Average Percentage of Blocks Covered*
APDC	*Average Percentage of Decisions Covered*
APFD	*Average Percentage of Faults Detected*
$APFD_c$	*Cost-cognizant Average Percentage of Faults Detected*
APSC	*Average Percentage of Statements Covered*
CE	*Coverage Effectiveness*
EVOMO	*Evolution-aware Economic Model for Regression Testing*
HMFD	*Harmonic Mean of the Rate of Fault Detection*
NAPFD	*Normalized Average Percentage of Faults Detected*
PCT	Priorização de Casos de Teste
PPGCA	Programa de Pós-Graduação em Computação Aplicada
RP	*Relative Position*
SIR	*Software-artifact Infrastructure Repository*
UTFPR	Universidade Tecnológica Federal do Paraná
WGFD	*Weighted Gain of Fault Detection*
WPFD	*Weighted Percentage of Failures Detected*

Capítulo 1

Introdução

Este capítulo inicia-se com uma breve contextualização do ambiente de pesquisa em que este trabalho se insere. Em seguida, são discutidas as motivações para este trabalho e quais os objetivos que pretende-se alcançar com o desenvolvimento do mesmo. Então, as contribuições deste trabalho são apresentadas e por fim, a organização adotada neste documento é brevemente descrita.

1.1 Contexto

A execução de um programa de computador com o objetivo de encontrar defeitos, atividade esta conhecida como teste, tem um papel muito importante no desenvolvimento de programas. Dentre as estratégias de teste descritas na literatura, o teste de caixa-preta é conhecido como o confrontamento entre o comportamento do programa em execução e a sua especificação, sem que o testador executando o teste conheça o funcionamento interno do programa ou como ele foi estruturado (MYERS; SANDLER; BADGETT, 2011).

O teste de sistema, outro nome dado ao teste de caixa-preta, é usado para projetar casos de teste em que são fornecidas entradas e são avaliadas as saídas geradas pelo programa. Esta estratégia tem como objetivo verificar se o programa está em conformidade com a sua especificação e avaliá-lo segundo o ponto de vista do usuário (DELAMARO; MALDONADO; JINO, 2007).

Segundo uma pesquisa realizada em 2015 com 1560 entrevistados em 32 países, o porcentual do orçamento para desenvolvimento alocado naquele ano para garantia de qualidade e testes foi, em média, de 35%. Além disso, 61% dos entrevistados indicaram como uma das prioridades desta área a diminuição do tempo gasto para liberar os produtos para o mercado (CAPGEMINI; HEWLETT-PACKARD; SOGETI, 2015).

Em busca da redução do tempo gasto na fase de testes, alternativas como o uso de testes automatizados têm aumentado consideravelmente nos últimos anos. Este tipo de teste é feito por meio de programas criados unicamente com o objetivo de testar o programa sendo desenvolvido. Como a execução destes é mais rápida que a dos testes manuais, no qual um testador executa o programa e valida os seus resultados, os testes automatizados tem sido muito importantes para dar mais agilidade à atividade de teste (RAFI et al., 2012).

De acordo com a mesma pesquisa, os testes automatizados respondiam em 2015 por 45% dos testes realizados nas empresas entrevistadas contra 28% em 2014. E entre os benefícios associados a eles, foram citados: a melhoria na detecção de falhas, o melhor controle

e transparência da atividade de testes, a redução no tempo utilizado para o ciclo de testes e a redução nos custos com testes (CAPGEMINI; HEWLETT-PACKARD; SOGETI, 2015).

Entretanto, somente o uso de testes automatizados pode não alcançar a redução no tempo necessário para o ciclo de testes ou nos custos envolvidos com a atividade de teste. Por isso, passaram a ser estudadas também formas de otimizar como esses testes são executados e entre elas está a priorização de casos de teste (YOO; HARMAN, 2012).

A priorização de casos de teste (PCT) propõe o ordenamento dos casos de teste de acordo com algum critério antes da sua execução. Ela tem como propósito aumentar a probabilidade de que um determinado objetivo seja alcançado mais rapidamente na ordem proposta do que em alguma outra ordem de execução dos casos de teste (ROTHERMEL et al., 1999).

A PCT pode melhorar o custo-benefício dos testes fazendo com que os testes mais importantes sejam executados mais cedo no ciclo de teste. E também pode fornecer um retorno mais rápido aos testadores e gerentes do projeto sobre a qualidade do programa, além de permitir aos desenvolvedores que iniciem a correção das falhas mais cedo. Além disso, caso seja bem sucedida, ela aumenta a probabilidade de que os testes mais importantes tenham sido realizados caso o ciclo de testes seja encerrado prematuramente (DO et al., 2010).

Desde a primeira vez em que a priorização foi estudada por Wong et al. (1997), diferentes abordagens já foram propostas e avaliadas na literatura. Dentre os critério mapeados por Catal e Mishra (2013) em estudos sobre a PCT, tem-se os baseados em cobertura, em distribuição, em decisão humana, em dados históricos, em requisitos, em modelos, em custos e os probabilísticos.

Destes critérios, a cobertura de código foi utilizada em 40% dos estudos, seja como parte integrante do critério para a abordagem sendo proposta, ou como critério base para comparação com os resultados obtidos pela abordagem sendo estudada (CATAL; MISHRA, 2013). Entretanto, nenhum dos critérios propostos baseados em cobertura de código encontrados na literatura utilizou o próprio código-fonte dos casos de teste de sistema, como proposto neste trabalho.

Neste trabalho propõe-se o uso dos grafos de chamadas dos casos de teste de sistema automatizados como critério para a priorização deste tipo de caso de teste. Estes grafos de chamadas são gerados estaticamente a partir da análise do código-fonte dos casos de teste automatizados e permitem que estes sejam priorizados sem a necessidade de nenhuma informação adicional ou execução prévia.

Para validar a abordagem proposta, dois estudos de caso foram executados. No primeiro estudo optou-se por selecionar uma das aplicações disponíveis em um repositório *online* chamado *Software-artifact Infrastructure Repository* (SIR) e que tem sido regularmente utilizado em estudos sobre priorização (CATAL; MISHRA, 2013). Já no segundo, uma aplicação desenvolvida em ambiente comercial utilizando falhas encontradas em tempo de projeto e casos de teste desenvolvidos durante o ciclo de testes da aplicação foi utilizada.

Para avaliar os resultados obtidos com a abordagem proposta em relação a outras ordens de execução dos casos de teste em cada um dos estudos de caso, a média harmônica da taxa de detecção de falhas em relação ao número de casos de teste executados foi utilizada. Esta métrica, conhecida como APFD (*Average Percentage of Faults Detected*), foi escolhida por ser até o ano de 2011 a métrica mais utilizada em estudos sobre PCT (CATAL; MISHRA, 2013).

1.2 Motivação

Diferentemente dos casos de teste automatizados para outras estratégias, como por exemplo a estratégia de caixa branca que inclui testes unitários e de integração, os casos de teste de sistema automatizados possuem código-fonte independente da aplicação sendo testada. E, por isso, podem ser priorizados sem a necessidade de nenhuma informação sobre a aplicação sendo testada, o que deve simplificar a sua utilização pela comunidade.

Este critério foi escolhido pois especula-se que o uso da cobertura de código dos casos de teste por meio dos grafos de chamadas possa obter resultados positivos de forma similar aos obtidos com a cobertura de código pelas diversas técnicas já propostas na literatura. Todavia, com a vantagem de não depender de relacionamentos entre os casos de teste e outras fontes de informação.

1.3 Objetivos

O objetivo geral deste trabalho é propor e avaliar uma nova abordagem para a priorização dos casos de teste de sistema automatizados. E, a fim de garantir que este objetivo seja alcançado e que esta abordagem para priorização dos casos de teste de sistema automatizados seja bem avaliada, os seguintes objetivos específicos foram definidos:

1. Avaliar se é possível aumentar a taxa de detecção de falhas dos casos de teste de sistema automatizados utilizando grafos de chamadas como critério de priorização.

2. Avaliar como a abordagem proposta se comporta em relação a outras abordagens já estudadas na literatura quando aplicadas ao mesmo cenário.

3. Avaliar também como a escolha do algoritmo usado na priorização impacta os resultados obtidos com esta nova abordagem.

1.4 Contribuições

Com base nos resultados obtidos em ambos os estudos de caso propostos utilizando a métrica APFD, concluiu-se que a abordagem proposta é capaz de alcançar resultados superiores a execução não ordenada dos casos de teste. E também, em relação a outras abordagens já estudadas na literatura, verificou-se que a abordagem proposta é capaz de alcançar resultados similares. Desta forma, pode-se dizer que o uso da abordagem proposta mostrou-se vantajosa na melhora da taxa de detecção de falhas de um conjunto de casos de teste de sistema automatizados.

Entretanto, devido as características desta pesquisa e dos seus estudos de caso, ainda não é possível generalizar os resultados obtidos para qualquer cenário em que casos de teste de sistema automatizados sejam utilizados. É importante notar que ambos os estudos de caso executados testaram aplicações desenvolvidas em Java seguindo os preceitos do paradigma de programação orientado a objetos.

Além disso, também os casos de teste automatizados em ambos os estudos de caso fizeram uso da linguagem Java. Desta forma, outros projetos utilizando linguagens diferentes ou outros paradigmas de programação podem obter resultados diferentes com o uso da abordagem

proposta. Portanto, estas características podem ser consideradas limitações e deixam em aberto lacunas de pesquisa para trabalhos futuros.

1.5 Organização do Texto

Dando sequência a este documento, no Capítulo 2 é apresentada a fundamentação teórica em que este trabalho se ampara, além de ser feita também uma análise dos trabalhos encontrados na literatura que apresentam alguma correlação com a abordagem proposta. A seguir, no Capítulo 3 é apresentado o método de pesquisa utilizado no desenvolvimento deste projeto. No Capítulo 4 é descrita a abordagem proposta por este trabalho e depois, no Capítulo 5 são apresentados os estudos de caso executados para avaliá-la. Por fim, no Capítulo 6 são revisitados os resultados obtidos neste trabalho e são analisadas suas correlações com os objetivos propostos, as suas implicações para esta área de pesquisa e as lacunas para futuras pesquisas.

Capítulo 2

Fundamentação Teórica

Afim de propiciar um melhor entendimento deste trabalho, este capítulo apresenta a fundamentação teórica em que se ampara este trabalho. Na Seção 2.1 são identificados os conceitos relevantes para este trabalho na área de priorização de casos de teste. A seguir, na Seção 2.2 são descritos os conceitos sobre teste automatizado utilizados neste documento e, por fim, os trabalhos correlatos a este são identificados na Seção 2.3. Fechando este capítulo, a Seção 2.4 apresenta as considerações finais.

2.1 Priorização de casos de teste

A técnica de priorização de casos de teste baseia-se na execução ordenada dos casos de teste a partir de algum critério. E o seu propósito é aumentar as chances que um determinado objetivo seja alcançado (ROTHERMEL et al., 1999).

Segundo Elbaum, Malishevsky e Rothermel (2000), o problema da priorização de casos de teste pode ser enunciado da seguinte maneira:

$$\begin{array}{ll} \textit{Dado}: & T\text{, um conjunto de casos de teste; } PT\text{, o conjunto de permutações de } T\text{;} \\ & f\text{, uma função de } PT \text{ para número reais.} \\ \textit{Problema}: & \text{Encontrar } T' \in PT \text{ tal que } (\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]. \end{array}$$

Para descrever os avanços na área de priorização, os trabalhos encontrados na literatura são aqui divididos com base em 4 categorias diferentes: critério utilizado para priorização, algoritmo utilizado para priorização, métrica usada para avaliação dos resultados e aplicação usada para avaliar a abordagem. E também faz-se uma diferenciação entre as diferentes origens de falhas usadas nos estudos desta área de pesquisa.

2.1.1 Critérios utilizados para a priorização

Uma das partes essenciais para a definição de uma abordagem para a PCT está na seleção do critério a ser utilizado no ordenamento dos casos de teste (função f da equação). Entre os critérios já utilizados em estudos nesta área, a cobertura de código da aplicação aparece como o critério mais utilizado, tendo sido usado em 46% dos estudos sobre priorização (ver Apêndice A).

Durante a execução dos casos de teste, uma das formas de verificar que partes da aplicação estão sendo testadas se dá por meio de medidas de cobertura de código. Normalmente, a quantidade de código coberta por um caso de teste é medida com a instrumentação estática do código da aplicação sendo testado. Durante a compilação da aplicação, as ferramentas responsáveis por efetuar essa medição inserem código instrumentado dentro do arquivo executável e este código instrumentado é então utilizado na execução dos casos de teste possibilitando que contadores internos gravem quais partes do código foram executadas (TIKIR; HOLLINGSWORTH, 2002).

As medidas de cobertura de código podem ser feitas usando-se diferentes granularidades e entre as medidas mais comuns tem-se a cobertura por instruções, por ramos de decisão e por função. Quando uma aplicação é medida em relação a cobertura de código por instruções executadas, cada linha de código é reconhecida como uma unidade diferenciada e ao medir a cobertura alcançada tem-se exatamente quais instruções foram ou não executadas. Já na cobertura por ramos de decisão, são medidos apenas quais caminhos foram executados em relação as instruções de decisão implementadas no código da aplicação. Por fim, para a cobertura por função são medidas apenas que funções/métodos implementados no código da aplicação foram executados (LI; HARMAN; HIERONS, 2007).

Entretanto, estas medidas de cobertura citadas necessitam que a aplicação seja executada uma ou mais vezes para que os dados de cobertura sejam obtidos antes que os casos de teste possam ser priorizados utilizando-se estes dados. Uma forma de evitar isto, se dá pelo uso dos grafos de chamadas obtidos por meio da análise estática do código-fonte. Neste caso, o código da aplicação é analisado sem a necessidade de ser executado previamente.

Os grafos de chamadas são representações estáticas das invocações dinâmicas entre métodos de uma aplicação. Cada nó do grafo de chamadas representa um método da aplicação e as arestas indicam cada uma das possíveis chamadas de função entre os diferentes métodos (HALL; KENNEDY, 1992). Na Figura 2.1 é exemplificado como o grafo de chamadas seria criado para uma aplicação hipotética.

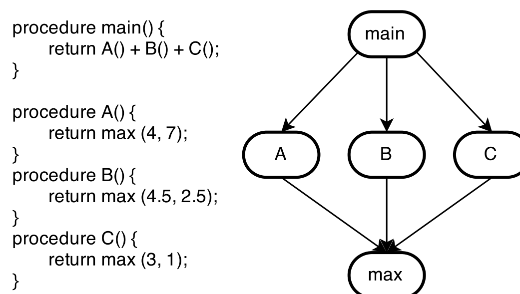


Figura 2.1: Exemplo de um grafo de chamadas. Fonte: Grove et al. (1997)

Além dos critérios baseados em cobertura de código, outro critério estudado em alguns trabalhos foi o potencial de exposição de falhas. Para calcular este valor potencial de cada caso de teste, mutantes da aplicação são gerados e com base na quantidade de mutantes que cada caso de teste consegue matar é gerada uma estimativa do seu potencial em encontrar falhas (ROTHERMEL et al., 1999).

Também os dados históricos de execução dos casos de teste foram utilizados em diferentes critérios propostos para a PCT. Entre estes critérios, tem-se por exemplo, o uso do histórico de execução para priorizar os casos de teste que tenham detectado falhas nas versões

mais recentes da aplicação sendo testada (LIN et al., 2013). Além desse critério, outras informações históricas como o tempo de execução (MARIJAN; GOTLIEB; SEN, 2013), a gravidade das falhas encontradas (PARK; RYU; BAIK, 2008) e o histórico das alterações usado no cálculo do impacto das mudanças (SHERRIFF; LAKE; WILLIAMS, 2007) também foram estudadas.

Os requisitos da aplicação também foram utilizados como critério em diferentes estudos para a PCT. Algumas das informações relativas aos requisitos da aplicação usados na priorização foram a prioridade dada pelo cliente, a volatilidade do requisito e a sua complexidade de implementação (SRIKANTH; BANERJEE, 2012).

Outras abordagens ainda se basearam em modelos do sistema para priorizar os casos de teste com base na análise da cobertura destes modelos. Alguns dos modelos utilizados foram: máquinas de estados estendidas (KOREL; TAHAT; HARMAN, 2005), oráculos de teste (STAATS; LOYOLA; ROTHERMEL, 2012) e dados de comunicação entre Web Services (MEI et al., 2011).

Além dos critérios já citados, os dados de entrada dos casos de teste também foram estudados como critério para a PCT, como por exemplo, o uso da localização geográfica do usuário (ZHAI; JIANG; CHAN, 2014). E nos estudos em que a PCT foi aplicada a iteração combinatorial, os próprios vetores de cobertura gerados para testar o sistema foram usados na sua priorização (BRYCE; COLBOURN, 2006).

2.1.2 Algoritmos utilizados para a priorização

Para priorizar o conjunto de casos de teste, vários algoritmos já foram estudados. Entre eles, destacam-se os algoritmos gulosos que foram usados em 82% dos estudos mapeados (ver Apêndice A). Entre os algoritmos gulosos, duas variações tem sido usadas com maior frequência: guloso total e guloso adicional.

Os algoritmos gulosos caracterizam-se pelo uso de uma função de priorização que permite selecionar os casos de teste mais relevantes para execução com base no critério sendo usado. Tomando como exemplo o critério de cobertura de código, um algoritmo guloso prioriza os casos de teste com maior cobertura de código para serem executados primeiramente.

A diferença entre os algoritmos gulosos total e adicional está na quantidade de vezes que a prioridade de cada caso de teste é calculada. No caso do algoritmo guloso total, a prioridade de cada caso de teste é calculada uma única vez e depois disso, os casos de teste são ordenados de forma a priorizar a execução dos casos de teste mais relevantes antes. Já no algoritmo guloso adicional, a prioridade de todos os casos de teste que ainda não foram priorizados é recalculada a cada iteração a fim de considerar a cobertura adicional que cada caso de teste irá incrementar ao conjunto dos casos de teste já priorizados. Desta forma, o algoritmo guloso adicional garante que o próximo caso de teste a ser escolhido sempre maximiza o resultado naquela iteração (ROTHERMEL et al., 1999).

Além dos algoritmos gulosos, outro grupo de algoritmos estudados para PCT foram os algoritmos de busca local, entre eles, *Hill Climbing* (LI; HARMAN; HIERONS, 2007) e *Alternative Variable Method* (WANG et al., 2014).

Também os algoritmos evolucionários foram estudados para PCT, entre eles, os algoritmos genéticos (LI; HARMAN; HIERONS, 2007), de otimização por enxame de partículas (HLA; CHOI; PARK, 2008), de otimização por colônia de formigas (CHEN et al., 2009) e o algoritmo evolucionário $(1+1)$ (WANG et al., 2014).

Outro grupo de algoritmos estudado foram os algoritmos de aprendizado de máquina. Neste grupo temos as redes Bayesianas (MIRARAB; TAHVILDARI, 2008), as redes neurais (BELLI; EMINOV; GOKCE, 2007), a análise de *clusters* (LEON; PODGURSKI, 2003) e o ranqueamento baseado em casos (TONELLA; AVESANI; SUSI, 2006).

Por fim, além dos algoritmos citados nos grupos anteriores, alguns outros algoritmos também foram estudados para PCT. São eles: *Beam search* (JIANG; CHAN, 2013), *Fielder ordering* (RAMANATHAN et al., 2008), *Simulated annealing* (KAUFFMAN; KAPFHAMMER, 2012), *Adaptive random testing* (JIANG et al., 2009) e a força bruta (permutação de todas a possibilidades para um dado conjunto de casos de teste) (LIMA et al., 2009).

2.1.3 Métricas usadas para avaliação dos resultados

Para medir os resultados obtidos com diferentes abordagens para priorização e poder compará-los, diferentes métricas já foram usadas em estudos sobre PCT. Dentre estas métricas, a APFD (do inglês, *Average Percentage of Faults Detected*) foi a mais utilizada, tendo sido usada em 41% dos estudos mapeados (ver Apêndice A).

Esta métrica é calculada a partir da média ponderada do percentual de falhas detectadas e ela indica quão rapidamente um conjunto de casos de teste priorizado é capaz de detectar as falhas presentes na aplicação sendo testada. Os valores calculados por ela variam de 0 a 100 e valores maiores indicam que as falhas são detectadas mais rapidamente.

Segundo Elbaum, Malishevsky e Rothermel (2002), sendo T o conjunto de n casos de teste disponíveis para execução e F o conjunto de m falhas presentes na aplicação sendo testada. E sendo TF_i o primeiro caso de teste da ordem T' a detectar uma falha i. O valor da APFD para o conjunto de teste T' é dado pela equação:

$$APFD(T') = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

Para exemplificar a forma como a APFD é calculada, considere que uma aplicação possui 10 falhas em uma dada versão e 5 casos de teste são utilizados para testá-la. A Tabela 2.1 apresenta a capacidade de detecção das falhas (1 a 10) de cada caso de teste (A a E).

Tabela 2.1: Casos de teste e lista de falhas expostas. Fonte: Elbaum, Malishevsky e Rothermel (2000)

Caso de Teste	Falha									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

Supondo que os casos de teste sejam colocados na ordem **A-B-C-D-E** para formar um conjunto priorizado T_1. A Figura 2.2 mostra o porcentual de falhas detectadas versus o percentual de T_1 executado. Depois de executar o caso de teste **A**, 2 das 10 falhas são detectadas, portanto 20% das falhas foram detectadas após 20% de T_1 ter sido executado. Após executar o caso de teste **B**, mais 2 falhas são detectadas e portanto 40% das falhas foram detectadas após

40% de T_1 ter sido executado. A área abaixo da curva no gráfico representa a média ponderada do percentual de falhas detectadas durante o ciclo de execução do conjunto de teste. Assim sendo, esta área é igual a APFD para o conjunto de casos de teste priorizado T_1 que, neste exemplo, é de 50%.

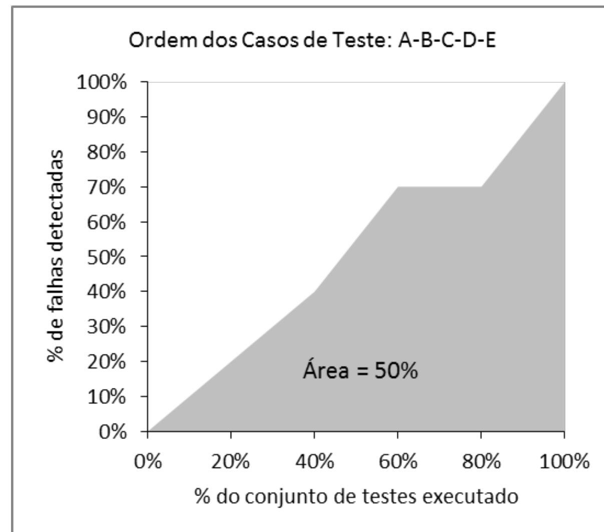


Figura 2.2: APFD para conjunto de teste priorizado T_1. Fonte: Elbaum, Malishevsky e Rothermel (2000)

Na Figura 2.3 a ordem de execução do conjunto de casos de teste (T_2) é alterada para **E-D-C-B-A**, o que causa uma detecção mais rápida das falhas do que em T_1 com uma APFD de 64%. Já na Figura 2.4 é apresentado o melhor ordenamento possível para este conjunto de casos de teste (T_3). A ordem de execução **C-E-B-A-D** gera uma APFD igual a 84%.

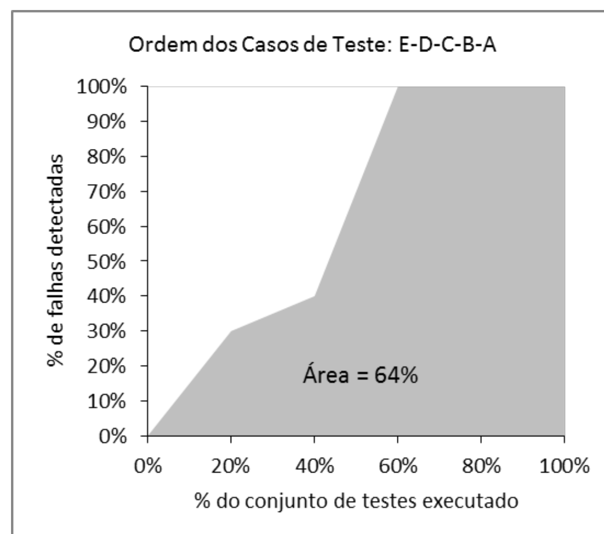


Figura 2.3: APFD para conjunto de teste priorizado T_2. Fonte: Elbaum, Malishevsky e Rothermel (2000)

Baseando-se na APFD, outras métricas foram propostas a partir de modificações no cálculo da métrica. Na $APFD_c$, por exemplo, a fórmula de cálculo da métrica é modificada para

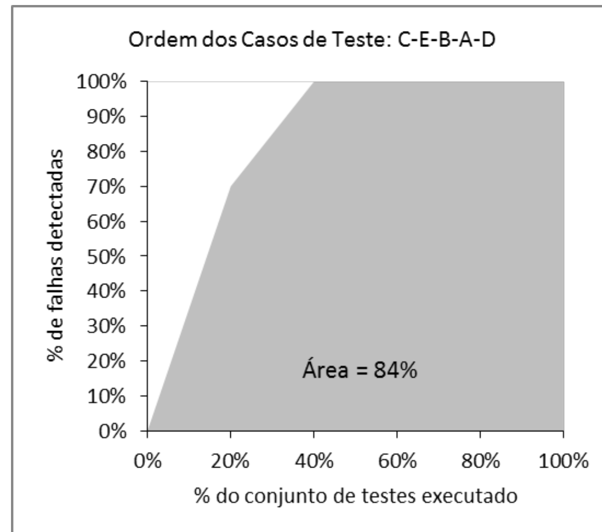


Figura 2.4: APFD para conjunto de teste priorizado T_3. Fonte: Elbaum, Malishevsky e Rothermel (2000)

que seja levada em consideração o custo de execução dos casos de teste e a gravidade das falhas encontradas (ELBAUM; MALISHEVSKY; ROTHERMEL, 2001).

De maneira semelhante, na métrica WPFD (do inglês, *Weighted Percentage of Failures Detected*) considera-se apenas a gravidade das falhas encontradas em relação aos requisitos do sistema sem levar em conta o custo de execução dos casos de teste (SRIKANTH; BANERJEE, 2012). Já a WGFD (do inglês, *Weighted Gain of Fault Detection*) permite que seja definido um peso para o ganho obtido com a rápida detecção das falhas (LV; YIN; CAI, 2013).

Na métrica HMFD (do inglês, *Harmonic Mean of the rate of Fault Detection*) calcula-se a média harmônica da taxa de detecção de falhas (ZHAI et al., 2010). E na NAPFD (do inglês, *Normalized Average Percentage of Faults Detected*), considera-se o número de casos de teste executados em relação ao número total de casos de teste e o número de falhas que não foram encontradas ao término da execução para calcular o seu valor (QU; COHEN; WOOLF, 2007).

Além das métricas baseadas na APFD, outras métricas foram propostas com base na detecção de falhas do conjunto de casos de teste priorizados. Entre elas temos, a quantidade de falhas não encontradas (CARLSON; DO; DENTON, 2011), a taxa de detecção de falhas (SAMPATH et al., 2008), o número total de falhas detectadas (XU; DING, 2010) e a proporção do conjunto de casos de teste necessária para detectar todas as falhas (MARIANI; PAPAGIANNAKIS; PEZZÈ, 2007).

Na métrica RP (do inglês, *Relative Position*) calcula-se a posição relativa mais provável em que cada uma das falhas será encontrada com a abordagem sendo proposta. Isto evita que abordagens com múltiplas soluções para um mesmo conjunto de teste, por exemplo pelo uso de funções randômicas para desempate no ordenamento dos casos de teste, sejam avaliadas apenas por uma única execução (KOREL; TAHAT; HARMAN, 2005).

A fim de evitar a necessidade de usar falhas reais ou semear falhas para calcular métricas como a APFD, a métrica CE (do inglês, *Coverage Effectiveness*) permite que uma função de cobertura cumulativa dos requisitos sendo avaliados seja utilizada para calcular a taxa de cobertura destes requisitos durante a execução do conjunto de casos de teste (KAPFHAMMER; SOFFA, 2007).

Para medir a eficiência dos algoritmos de priorização, a cobertura de código alcançada após a priorização também foi utilizada (AGGRAWAL; SINGH; KAUR, 2004), além de outras métricas baseadas na cobertura de código. Entre elas, a APBC (do inglês, *Average of the Percentage of Blocks Covered*), a APDC (do inglês, *Average of the Percentage of Decisions Covered*) e a APSC (do inglês, *Average of the Percentage of Statements Covered*) (LI; HARMAN; HIERONS, 2007).

Outra forma utilizada ainda para medir a eficiência das técnicas de PCT é através do uso do tempo de execução e métricas associadas a ele. Entre estes usos temos: o tempo necessário para a priorização dos casos de teste (JONES; HARROLD, 2003) e o tempo necessário para detectar a primeira falha (ELBAUM; ROTHERMEL; PENIX, 2014).

Para avaliar o custo-benefício envolvido com a utilização de abordagens para priorização, alguns modelos econômicos também foram propostos. Um deles foi proposto por Malishevsky, Rothermel e Elbaum (2002) e nele são considerados os custos com análise, com a priorização e com o tempo de espera para a detecção das falhas. Já no EVOMO (do inglês, *EVOlution-aware economic MOdel for regression testing*) que estendeu o modelo anterior, foram incorporados os custos com configuração do ambiente, detecção e correção dos casos de teste obsoletos, execução dos casos de teste, validação dos resultados e também o custo associado ao atraso na detecção das falhas (DO; ROTHERMEL, 2008).

2.1.4 Origem das falhas usadas para avaliar as abordagens

Para que métricas baseadas em falhas encontradas nas aplicações sejam utilizadas na avaliação de abordagens propostas para a PCT, estas falhas devem estar presentes na aplicação sendo usada no experimento. E elas podem ser obtidas de duas formas diferentes, por meio de falhas reais ou falhas semeadas.

As falhas são ditas reais quando encontradas durante o desenvolvimento da aplicação e disponibilizadas junto com ela para a execução do experimento desejado. Já as falhas semeadas, são criadas a partir da versão correta da aplicação usando-se uma de duas técnicas: manualmente ou por mutação.

No caso da falhas semeadas manualmente, as falhas são inseridas intencionalmente no código por um humano, simulando possíveis erros que poderiam ser cometidos pelos desenvolvedores da aplicação. Já no caso das falhas semeadas por mutação, variações do código original são geradas a partir da aplicação de operadores no código (conhecidos como operadores de mutação). E estas variações, também conhecidas como mutantes, são usadas como versões falhas da aplicação.

Estudos comparando o uso de falhas semeadas (manualmente e por meio de mutações) e falhas reais com técnicas de priorização, mostraram que o uso de falhas semeadas pode obter resultados semelhantes as falhas reais. E desta forma, na ausência de falhas reais, falhas semeadas podem ser usadas sem prejudicar os resultados obtidos na experimentação de abordagens de priorização (ANDREWS; BRIAND; LABICHE, 2005; DO; ROTHERMEL, 2006).

2.1.5 Aplicações usadas para avaliar as abordagens

Para auxiliar a comunidade científica na obtenção de aplicações para experimentação com teste de software, repositórios tem sido criados para disponibilizar aplicações a serem usadas nestes experimentos. No caso dos estudos sobre priorização, um repositório que se

destaca é o *Software-artifact Infrastructure Repository* (SIR), tendo sido utilizado em 1 de cada 5 estudos mapeados (ver Apêndice A).

Esta grande utilização pode ser atribuída ao fato do SIR disponibilizar diversas aplicações em diferentes linguagens de programação, muitas vezes com mais de uma versão para cada uma delas e também com falhas disponíveis para cada uma dessas versões. Simplificando assim a execução de experimentos com novas abordagens para priorização (DO; ELBAUM; ROTHERMEL, 2005).

O primeiro objeto de estudo usado para avaliar abordagens de priorização foi o *space* (WONG et al., 1997). Ele foi desenvolvido pela agência espacial europeia e foi utilizado em quase 10% dos estudos mapeados (ver Apêndice A).

Outro conjunto de aplicações a ser utilizado logo no início dos estudos sobre priorização foram as aplicações desenvolvidas por uma equipe de pesquisa da Siemens (ROTHERMEL et al., 1999). Este conjunto de aplicações, comumente intitulado de *siemens suite*, foi utilizado em 12% dos estudos mapeados (ver Apêndice A).

Alguns autores utilizaram aplicações desenvolvidas dentro das próprias universidades (BRYCE; MEMON, 2007) ou modelos de aplicações para os estudos de caso (BAI; LAM; LI, 2004). Em outros casos, aplicações comerciais foram utilizadas para o estudo de abordagens para PCT (SRIKANTH; COHEN; QU, 2009) ou ainda algumas aplicações web disponibilizadas em repositórios abertos (MEI et al., 2011).

2.2 Teste automatizado

Segundo Dustin, Rashka e Paul (1999), a automação de testes pode significar a automação de qualquer uma das atividades envolvidas no teste de uma aplicação; incluindo o desenvolvimento, execução dos casos de teste, a verificação de requisitos de teste e o uso de ferramentas automatizadas de teste. Entre as razões para o uso de automação incluem-se o fato de testes manuais demandarem tempo, e a possibilidade de aumentar a eficiência dos testes ao permitir que os testes sejam executados interativamente após alterações na aplicação serem efetuadas.

Estudos recentes confirmam esta afirmação ao indicar que entre os maiores benefícios percebidos com a automação de testes está a redução no tempo necessário para execução dos testes. Além disso, outros benefícios apontados são a melhoria na qualidade do produto sendo desenvolvido, aumento na cobertura de código alcançada pelos testes e um aumento na capacidade de detectar falhas (RAFI et al., 2012).

Apesar da PCT não estar necessariamente ligada a execução de casos de teste automatizados, podendo ser utilizada também para execução de casos de teste manuais, nota-se na literatura uma forte tendência a sua aplicação em conjuntos de casos de teste automatizados. Por exemplo, em nenhum dos trabalhos mapeados fica explícito o uso de testes manuais no estudo das abordagens propostas (ver Apêndice A).

Segundo Delamaro, Maldonado e Jino (2007), a atividade de testes pode ser dividida em três fases: testes de unidade, testes de integração e testes de sistema. Testes de unidade visam verificar o funcionamento das menores unidades de uma aplicação, como por exemplo, funções e procedimentos; testes de integração visam verificar o funcionamento das diversas partes de uma aplicação juntas; e testes de sistema visam verificar que as funcionalidades especificadas para aplicação estão corretamente implementadas.

Em relação à automação da execução dos casos de teste, estas três fases da atividade de teste também possuem características diferenciadas, principalmente em relação ao acesso por parte dos casos de teste ao código-fonte da aplicação. Enquanto os testes de unidade e os testes de integração tem acesso direto ao código da aplicação para testá-la, os testes de sistema tem acesso apenas as interfaces visíveis ao usuário para verificar o correto funcionamento da aplicação.

2.3 Trabalhos correlatos

Estudada pela primeira vez por Wong et al. (1997), a PCT fez parte da abordagem proposta utilizando a cobertura de código juntamente ao algoritmo guloso adicional para a priorização. Entretanto, neste estudo a priorização era executada somente entre os casos de teste previamente selecionados para execução baseando-se nas alterações no código e em dados de execuções anteriores.

Foram Rothermel et al. (1999) os primeiros a propor o estudo da PCT isoladamente e também a propor o uso da APFD como métrica para medir os resultados obtidos com a priorização. Ao estudar o uso dos critérios de cobertura de código por ramos e por instruções e do potencial de exposição de falhas utilizando os algoritmos guloso total e adicional, eles concluíram ser possível melhorar a APFD de um conjunto de casos de teste por meio da sua priorização.

Nos anos seguintes, diversos outros critérios foram estudados para a PCT. Por exemplo, dados históricos de execução (KIM; PORTER, 2002), dados de entrada para os casos de teste (YU; NG; CHAN, 2003), modelos do sistema (KOREL; TAHAT; HARMAN, 2005), requisitos do sistema (SRIKANTH; WILLIAMS; OSBORNE, 2005) e vetores de cobertura (BRYCE; COLBOURN, 2006). Todos conseguiram mostrar ganhos em relação à execução não ordenada dos casos de teste. Entretanto, quando comparados aos resultados obtidos utilizando-se a cobertura de código, poucos critérios conseguiram ser tão eficazes.

A primeira utilização dos grafos de chamadas como critério de priorização foi por meio de um índice calculado a partir dos grafos de chamadas dos casos de teste, dos dados sobre mudanças no código da aplicação e do histórico sobre a detecção de falhas dos casos de teste. Esta abordagem foi avaliada por meio da priorização dos casos de teste unitários de duas aplicações obtidas no SIR (*jtopas* e *xml-security*). E concluiu que o índice gerado com base nestes dados consegue melhorar a APFD para os conjuntos de casos de teste estudados (MA; ZHAO, 2008).

O segundo trabalho a utilizar os grafos de chamadas para a priorização fez uso exclusivamente deles como critério para a priorização. Entretanto, diferentemente da abordagem proposta aqui, ele priorizou casos de teste unitários usando seus grafos de chamadas, levando em consideração assim o código-fonte da aplicação sendo testada. Ao avaliar a abordagem priorizando os casos de teste para duas aplicações obtidas no SIR (*jtopas* e *ant*), ele alcançou resultados similares para a APFD aos obtidos com a cobertura de código por funções (ZHANG et al., 2009).

Além dos dois trabalhos citados acima, um terceiro trabalho encontrado na literatura propõe a utilização dos grafos de chamadas juntamente com os dados sobre as alterações no código da aplicação sendo testada a fim de priorizar os casos de teste com base na sua cobertura das alterações efetuadas no código. Entretanto, este trabalho não executa nenhum tipo de expe-

rimento para validar esta abordagem ou compará-la a outras abordagens propostas na literatura (ZHI-HUA; YONG-MIN; YING-AI, 2012).

Desta forma, pode-se afirmar que, apesar do critério utilizado na abordagem proposta neste trabalho já ter sido utilizado em trabalhos anteriores com sucesso, o cenário em que se propõe a sua utilização justifica o seu estudo para a evolução desta área de pesquisa. Enquanto os trabalhos anteriores tem utilizado os grafos de chamadas para priorizar casos de teste com base no código-fonte da aplicação, a abordagem proposta neste trabalho utiliza o mesmo critério para priorizar casos de teste cujo código-fonte independe da aplicação sendo testada.

2.4 Considerações finais

Neste capítulo foram definidos os conceitos sobre priorização de casos de teste e sobre teste automatizado que serão utilizados no decorrer deste documento. Também fez-se uma descrição da evolução da área de pesquisa em que este trabalho se insere, tendo como foco principal os trabalhos correlatos a este e possibilitando assim ao leitor uma melhor compreensão da maneira como este trabalho se compara aos trabalhos publicados anteriormente. No capítulo seguinte é apresentado o método de pesquisa utilizado durante o desenvolvimento deste projeto.

Capítulo 3

Método de Pesquisa

Este capítulo apresenta o método de pesquisa utilizado para desenvolver a proposta estudada neste trabalho e também as atividades definidas para avaliá-la. Na Seção 3.1 são descritas as atividades desenvolvidas durante a execução deste trabalho. E na Seção 3.2 são descritas cada uma das atividades desenvolvidas para avaliar a proposta sendo estudada. Por fim, na Seção 3.3 são feitas as considerações finais deste capítulo.

3.1 Sobre o método de pesquisa utilizado neste trabalho

A Figura 3.1 apresenta as atividades executadas para o desenvolvimento deste trabalho. Estas atividades foram executadas durante este projeto e basearam-se na metodologia de pesquisa proposta por Wazlawick (2014).

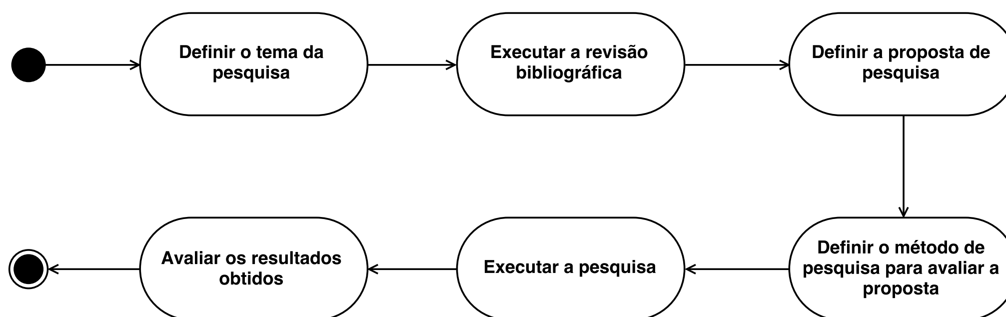


Figura 3.1: Diagrama de atividades para a execução deste trabalho

Após a leitura de alguns artigos envolvendo execução de teste automatizados, definiu-se que o tema da pesquisa deste trabalho seria a priorização de casos de teste. A partir desta definição, iniciou-se então a atividade de revisão da bibliografia para esta área de pesquisa.

Com o objetivo de obter uma visão atualizada desta área de pesquisa e possibilitar que lacunas fossem identificadas, optou-se pela execução de um estudo de mapeamento sobre priorização de casos de teste. Como resultado deste estudo, foram mapeados 178 artigos sobre PCT entre 1997 e 2014. O detalhamento do método utilizado neste estudo, assim como, os resultados detalhados obtidos por ele podem ser vistos no Apêndice A.

Com a revisão bibliográfica finalizada, uma lacuna identificada foi a ausência de estudos voltados aos casos de teste de sistema. E apesar de várias abordagens propostas serem aplicáveis neste contexto, nenhum estudo deixou claro que este cenário tenha sido avaliado e quais os resultados obtidos. Baseando-se nesta lacuna identificada com a revisão bibliográfica, definiu-se então a proposta de pesquisa desenvolvida neste trabalho que será apresentada no Capítulo 4.

Uma vez definida a proposta de pesquisa, iniciou-se a definição das atividades necessárias para avaliar esta proposta. Na Seção 3.2 é descrito o resultado obtido do desenvolvimento desta atividade.

A seguir, as atividades definidas no método de pesquisa para avaliar a proposta foram executadas e, por fim, os resultados obtidos com esta pesquisa foram avaliados. No Capítulo 5 são apresentados os resultados obtidos com a execução desta pesquisa e também são feitas as avaliações destes resultados.

3.2 Sobre o método de pesquisa utilizado para avaliar a proposta

Para avaliar se a abordagem proposta neste trabalho pode possibilitar uma melhora na taxa de detecção de falhas de um conjunto de casos de teste de sistema automatizados, dois estudos de caso foram definidos. A opção por dois estudos foi tomada com o objetivo de avaliar a abordagem proposta em relação a três cenários diferentes:

1. Comparação com a execução não ordenada dos casos de teste.
2. Comparação com uma abordagem tradicional de priorização por cobertura de código.
3. Avaliação do seu comportamento com uma aplicação retirada de um ambiente comercial.

Apesar destes três cenários não serem mutuamente excludentes, a aplicação comercial disponível para ser usada nesta pesquisa não possibilitava a aplicação da abordagem tradicional de priorização. Por se tratar de uma aplicação Web rodando em um ambiente complexo, a obtenção de informações sobre a cobertura de código da aplicação para cada caso de teste não era viável. Fez-se necessária portanto, a execução de dois estudos de caso independentes.

Desta forma, para o primeiro estudo de caso optou-se por selecionar um projeto já utilizado em outros estudos sobre priorização o que permitiu que os resultados obtidos com a abordagem proposta fossem comparados a uma abordagem tradicional de priorização, além de também permitir a comparação com os resultados da execução não ordenada dos casos de teste. E, no segundo estudo de caso, um projeto desenvolvido em ambiente comercial foi utilizado para validar como a abordagem proposta se compara à execução não ordenada dos casos de teste neste cenário.

3.2.1 Primeiro estudo de caso

Para execução do primeiro estudo de caso, as seguintes atividades foram definidas de forma a permitir que a abordagem proposta fosse avaliada e os objetivos desta pesquisa para

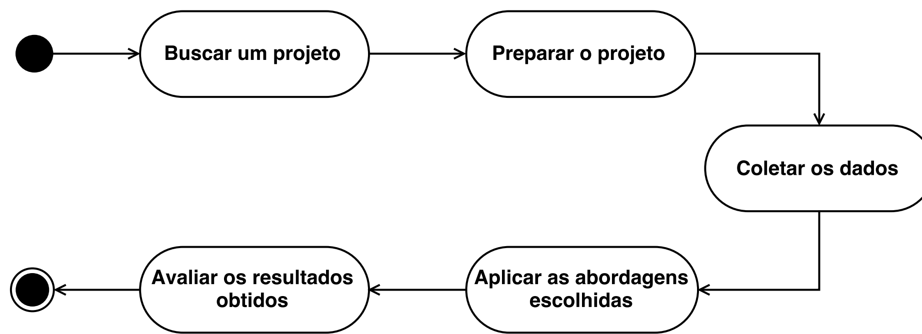


Figura 3.2: Diagrama de atividades do primeiro estudo de caso

este estudo fossem alcançados. Na Figura 3.2 são apresentadas as atividades definidas para este estudo.

Para dar início a este estudo de caso, buscou-se um projeto que pudesse ser utilizado para este fim. Para tal, algumas características importantes para o projeto foram levantadas e os seguintes pré-requisitos foram definidos:

- Disponibilizar o código-fonte da aplicação para avaliação dos resultados obtidos com a abordagem de priorização por cobertura de código.
- Disponibilizar um histórico de falhas encontradas na aplicação para permitir o cálculo da taxa de detecção de falhas com cada uma das abordagens.
- Disponibilizar casos de teste de sistema automatizados para permitir a priorização pela abordagem sendo proposta.

Tendo o projeto sido escolhido, as preparações necessárias foram feitas para permitir que a aplicação fosse testada por meio de casos de teste de sistema automatizados. Estas preparações visaram possibilitar que a abordagem proposta fosse aplicada aos casos de teste da aplicação escolhida.

Com estas duas fases iniciais de preparação finalizadas, a aplicação foi testada usando os casos de teste disponíveis e os dados necessários para aplicação das abordagens escolhidas foram coletados. Para alcançar os objetivos propostos e comparar a abordagem proposta com outras abordagens tradicionais de cobertura de código, os seguintes critérios para priorização foram avaliados neste estudo de caso:

- Cobertura de instruções de código da aplicação sendo testada.
- Cobertura dos grafos de chamadas dos casos de teste de sistema automatizados.

Portanto, a etapa de coleta de dados obteve informações relativas a cobertura de código por instruções para cada caso de teste executado e também os grafos de chamadas de cada um dos casos de teste. Além disso, as informações relativas à detecção de falhas para cada um dos casos de teste foi armazenada para que os resultados obtidos com cada uma das abordagens de priorização estudadas fossem avaliados.

Por fim, as abordagens sendo estudadas foram aplicadas e os conjuntos de casos de teste priorizados foram analisados utilizando-se a métrica APFD para comparação dos resultados obtidos com cada uma das abordagens. A fim de permitir a comparação dos resultados obtidos com diferentes algoritmos de priorização, decidiu-se que os critérios propostos fossem aplicados utilizando-se dois algoritmos: o guloso total e o guloso adicional.

Desta forma, possibilitou-se a comparação neste estudo de caso entre os resultados obtidos com quatro abordagens diferentes para priorização de casos de teste de sistema automatizados:

- Cobertura de instruções usando o algoritmo guloso total.
- Cobertura de instruções usando o algoritmo guloso adicional.
- Cobertura de grafos de chamadas usando o algoritmo guloso total.
- Cobertura de grafos de chamadas usando o algoritmo guloso adicional.

Além disso, para permitir a comparação da abordagem proposta com a execução não ordenada dos casos de teste, duas outras ordens de execução foram consideradas:

- Ordem original de execução dos casos de teste.
- Ordens randômicas de execução dos casos de teste.

Como a ordem original de execução dos casos de teste está diretamente ligada à ferramenta utilizada para execução dos mesmos, optou-se também pela utilização do resultado médio obtido com ordens randômicas de execução. Da mesma forma que em outros trabalhos similares nesta área de pesquisa, 10 ordens randômicas de execução foram geradas e o resultado médio da taxa de detecção de falhas foi usado para efeitos comparativos (ZHANG et al., 2009; MEI et al., 2012; ZHAI; JIANG; CHAN, 2014).

3.2.2 Segundo estudo de caso

Ao contrário do estudo de caso anterior, neste segundo estudo de caso o projeto a ser utilizado já foi previamente escolhido e portanto a quantidade de atividades a serem executadas foi reduzida, ignorando-se as fases iniciais de busca e preparação do projeto. Na Figura 3.3 são apresentadas as atividades definidas para este segundo estudo.

Figura 3.3: Diagrama de atividades do segundo estudo de caso

Como o projeto em questão já havia sido desenvolvido anteriormente, este estudo de caso iniciou-se com a coleta dos dados sobre o projeto. A fim de alcançar os objetivos propostos para este estudo e comparar a abordagem proposta com a execução não ordenada dos casos de teste, além de avaliar como a escolha do algoritmo de priorização impacta os resultados obtidos, as seguintes ordens de execução foram avaliadas neste estudo:

- Priorização por cobertura de grafos de chamadas usando o algoritmo guloso total.
- Priorização por cobertura de grafos de chamadas usando o algoritmo guloso adicional.
- Ordem original de execução dos casos de teste.
- Ordens randômicas de execução dos casos de teste.

Para avaliar estas ordens de execução, a coleta de dados neste estudo de caso obteve os grafos de chamadas para cada um dos casos de teste executados, assim como as informações a respeito da capacidade de detecção de falhas para cada um deles. Além disso, foi coletada também qual a ordem original de execução dos casos de teste neste projeto.

Em seguida, aplicaram-se as abordagem escolhidas para este segundo estudo e, por fim, analisaram-se os resultados obtidos com estas abordagens utilizando-se a métrica APFD. Da mesma forma que no primeiro estudo de caso, a utilização do valor médio para 10 execuções randômicas do conjunto de casos de teste e a ordem original de execução dos casos de teste foram usados como exemplos de resultados de execuções não ordenadas dos casos de teste.

3.2.3 Diferença entre a abordagem proposta e a execução dos estudos de caso

Apesar da aplicação da abordagem proposta neste trabalho indicar que os grafos de chamadas são gerados antes da execução dos casos de teste (ver Seção 4.1.4), a ordem destas atividades não foi a mesma nos estudos de caso executados para avaliá-la. Isto ocorre pois a avaliação desta proposta foi feita utilizando-se os dados históricos sobre a execução e a detecção de falhas dos casos de teste de projetos desenvolvidos anteriormente. Esta escolha permitiu que diferentes abordagens fossem aplicadas utilizando as informações coletadas dos projetos e que os resultados obtidos por elas fossem comparados.

Sendo assim, em vez de priorizar os casos de teste antes da sua execução, os estudos de caso executam os casos de teste para obter as informações sobre a sua capacidade de detectar falhas e só depois priorizam os casos de testes. Podendo assim estudar várias abordagens e analisar como cada uma delas se comporta em relação umas às outras.

Entretanto, pode-se afirmar que essa adaptação na forma de execução das abordagens estudadas não impacta os resultados obtidos nesta pesquisa. Pois, ela não modifica a forma como a priorização seria executada caso as abordagens sendo estudadas fossem usadas em tempo de projeto. E por isso, pode-se afirmar que os mesmos resultados seriam encontrados caso elas fossem aplicadas durante o desenvolvimento dos projetos sendo estudados.

3.3 Considerações finais

Neste capítulo foram apresentados o método de pesquisa utilizado para o desenvolvimento deste trabalho e também o método de pesquisa utilizado para avaliar a abordagem proposta por meio de dois estudos de caso. No capítulo seguinte, a abordagem proposta por este trabalho é detalhada.

Capítulo 4

Priorização de Casos de Teste de Sistema Automatizados

Neste capítulo são detalhadas as informações a respeito da abordagem proposta neste trabalho e as atividades executadas para implementá-la. Na Seção 4.1 são descritas as motivações para o estudo da abordagem proposta, da escolha do critério e dos algoritmos estudados e também os passos necessários para aplicar esta abordagem. Na Seção 4.2 são indicadas as ferramentas utilizadas para aplicação da abordagem proposta. E na Seção 4.3 são detalhados como foram implementados os algoritmos estudados neste trabalho. Por fim, na Seção 4.4 são apresentadas as considerações finais deste capítulo.

4.1 Abordagem proposta

A abordagem proposta neste trabalho baseia-se na técnica de priorização de casos de teste para aumentar a taxa de detecção de falhas de um conjunto de casos de teste de sistema automatizados. Para alcançar este objetivo, o critério de priorização selecionado foi grafos de chamadas e fazendo uso deste critério, dois algoritmos de priorização foram estudados: guloso total e guloso adicional.

4.1.1 Sobre a priorização neste cenário

Quando avaliam-se os problemas encontrados na fase de testes de sistema, é comum constatar que o tempo necessário para executar cada um dos casos de teste e consequentemente o conjunto completo dos casos de teste de sistema para uma aplicação é muitas vezes proibitivo para que entregas sejam feitas em curtos espaços de tempo. E apesar da automatização destes casos de teste agilizar a sua execução, não é incomum encontrar aplicações em que os testes de sistema levem dias para serem executados.

Num cenário como este, aumentar as chances de encontrar falhas presentes na aplicação mais rapidamente por meio de mudanças na ordem em que os casos de teste são executados pode permitir que correções sejam iniciadas mais cedo. E estas correções fazem com que o uso da priorização de casos de teste permita intervalos mais curtos entre o início dos testes, a detecção de falhas e o início das correções necessárias durante o desenvolvimento de uma aplicação.

Entretanto, ao avaliarem-se os estudos na área de priorização de casos de teste, verifica-se a falta de trabalhos que estudem o comportamento dos diferentes critérios de priorização para casos de teste de sistema. Este fato motiva este trabalho a propor uma abordagem para este cenário que possa trazer ganhos reais à taxa de detecção de falhas.

4.1.2 Sobre o critério utilizado na priorização

Apesar da falta de estudos na área de priorização utilizando testes de sistema, é possível deduzir que alguns critérios estudados com outros tipos de teste, como por exemplo os testes de unidade, possam alcançar resultados similares neste cenário. Entre eles, os critérios com base na cobertura de código da aplicação sendo testada, nos dados históricos de execução dos casos de teste e nos requisitos do sistema.

Entretanto, o uso destes critérios também traria as mesmas desvantagens já mencionadas na literatura, entre elas, a necessidade de guardar informações adicionais sobre cada caso de teste, como por exemplo, a quantidade de código coberta por ele, o histórico de detecção de falhas para cada caso de teste ou as informações a respeito da cobertura dos requisitos por cada um deles. Isto adiciona maior complexidade ao processo de execução e manutenção dos casos de teste.

Ao utilizar os grafos de chamadas dos casos de teste de sistema automatizados, esta abordagem usa os dados presentes nos cenários de teste automatizado para diferenciá-los e priorizá-los com base no tamanho destes casos de teste e também na cobertura hipotética alcançada por eles. Sem a necessidade de nenhuma informação adicional além do próprio código-fonte dos casos de teste.

E, diferentemente dos outros critérios já estudados, os estudos anteriores utilizando grafos de chamadas não tem aplicação neste cenário por tratarem de casos de teste unitários que testam diretamente o código-fonte da aplicação. E isto faz com que os grafos de chamadas representem a cobertura dos casos de teste dentro do próprio código-fonte da aplicação sendo testada.

Por sua vez, quando utilizados com casos de teste de sistema, estes grafos de chamadas passam a representar unicamente a cobertura de cada caso de teste dentro do próprio código-fonte criado para testar a aplicação. E a forma como os casos de teste são escritos pode influenciar diretamente como eles são priorizados.

4.1.3 Sobre os algoritmos utilizados na priorização

Ao escolher dois algoritmos para serem estudados, pretende-se verificar como a escolha do algoritmo de priorização pode influenciar nos resultados obtidos neste cenário. E a escolha dos algoritmos guloso total e guloso adicional justifica-se pelos bons resultados obtidos em outros estudos e pelas diferenças esperadas em relação ao tempo de execução e a seleção dos casos de teste em ambos os casos.

No caso do algoritmo guloso total, utiliza-se um algoritmo mais simples e mais rápido para dar prioridade a execução dos casos de teste com o maior número de nós no grafo de chamadas. A razão para o uso deste algoritmo é que ao selecionar os casos de teste com maior número de nós, estariam sendo selecionados os casos de teste mais complexos e que poderiam estar testando mais funcionalidades da aplicação.

Por sua vez com o algoritmo guloso adicional, utiliza-se um algoritmo mais complexo e mais lento para priorizar a execução dos casos de teste com o maior número de nós ainda não executados no grafo de chamadas. A razão para o uso deste algoritmo é que ao selecionar o maior número de nós não executados, estariam sendo selecionados os casos de teste que verificam funcionalidades diferentes da aplicação que ainda não foram testadas pelos casos de teste priorizados anteriormente.

Além das razões apresentadas acima, a escolha destes dois algoritmos gulosos baseia-se em resultados disponíveis na literatura. Estudos comparativos entre estes algoritmos gulosos e outras classes de algoritmos, como por exemplo algoritmos de busca local ou evolucionários, mostraram que a efetividade dos algoritmos gulosos na priorização de casos de teste, aliada a simplicidade de implementação destes algoritmos, justificam a sua utilização (LI; HARMAN; HIERONS, 2007; LI et al., 2010).

4.1.4 Sobre a aplicação da priorização

Para utilizar esta abordagem, os passos executados para efetuar a priorização são apresentados na Figura 4.1.

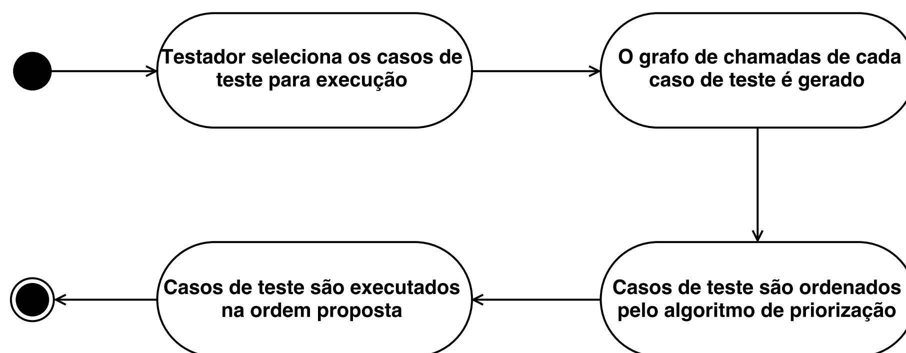


Figura 4.1: Diagrama de atividades com os passos executados para a priorização

Logo que o testador seleciona um conjunto de casos de teste de sistema automatizados para serem executados, inicia-se a geração dos grafos de chamadas para cada um dos casos de teste deste conjunto para que eles possam ser usados na priorização. Estes grafos de chamadas são gerados individualmente e representam as chamadas de função dentro do código-fonte dos próprios casos de teste utilizando funções e métodos que em algum momento interagem com a aplicação sendo testada.

Tendo sido gerados os grafos de chamadas, os casos de teste passam a ser ordenados com base nos seus grafos de chamadas levando em consideração o algoritmo de priorização sendo utilizado. E, ao final deste passo, tem-se a ordem de execução gerada por esta abordagem e os casos de teste podem então ser executados na ordem proposta.

4.2 Ferramentas auxiliares

Para possibilitar a aplicação da abordagem proposta, buscou-se na literatura ferramentas que pudessem gerar o grafo de chamadas para cada um dos casos de teste e obter os dados sobre cobertura de código da aplicação sendo testada.

4.2.1 Ferramenta para gerar o grafo de chamadas

Com base em artigos sobre construção estática do grafo de chamadas e do uso de ferramentas de busca, duas ferramentas foram encontradas:

- Soot[1] (LAM et al., 2011)
- WALA - T.J. Watson Libraries for Analysis[2]

Ao analisar ambas as ferramentas e fazer algumas simulações com casos de teste automatizados para gerar seus grafos de chamadas, a ferramenta Soot foi escolhida para ser usada nesta pesquisa. A motivação para esta escolha foi a possibilidade presente nesta ferramenta de escolher qual o método raiz a ser utilizado para a criação do grafo, facilitando assim a criação dos grafos de chamadas para cada um dos casos de teste.

No caso da ferramenta WALA, o método *main* da aplicação sempre é usado como raiz do grafo de chamadas. O uso desta ferramenta obrigaria o usuário a gerar um método *main* em tempo de execução apontando para o caso de teste ao qual se tem a necessidade de criar o grafo de chamadas. Isto dificultaria bastante o processo de criação destes grafos para cada caso de teste.

Além da ferramenta para criação dos grafos de chamadas, uma outra ferramenta auxiliar chamada de Averroes[3] foi utilizada. Quando executada antes da ferramenta que irá gerar os grafos de chamadas, ela permite que a criação do grafo de chamadas de uma aplicação seja simplificado ao ignorar as chamadas de função para bibliotecas usadas por ela. Isto permite que o grafo de chamadas seja mais simples, porém sem perder a representatividade das chamadas de função dentro da própria aplicação (ALI; LHOTÁK, 2013).

Entretanto, a escolha da ferramenta Soot para criação dos grafos de chamadas também traz uma limitação ao uso da abordagem proposta. Durante as simulações usando esta ferramenta, verificou-se que ela não conseguiria gerar os grafos de chamadas para os casos de teste devido a problemas no reconhecimento das anotações utilizadas por JUnit para indicar quais são os métodos de teste (anotações @*Test*).

Para mitigar esta limitação durante a pesquisa, estas anotações foram removidas do código antes da criação dos grafos de chamadas, alteração esta que não oferece ameaça à validade sobre os resultados obtidos. Entretanto, para a aplicação da abordagem proposta sem intervenção humana, seria vital que esta limitação fosse solucionada ou outra ferramenta com suporte a estas anotações fosse utilizada na criação dos grafos de chamadas.

[1] http://sable.github.io/soot/
[2] http://wala.sourceforge.net
[3] http://plg.uwaterloo.ca/ karim/projects/averroes/

4.2.2 Ferramenta para obter a cobertura de código

Para a aplicação da abordagem tradicional de priorização por cobertura de código, buscou-se também uma ferramenta que fizesse o cálculo de cobertura de código sobre a aplicação sendo testada após a execução de cada caso de teste. A partir dos próprios artigos sobre priorização lidos durante a revisão da literatura e do uso de ferramentas de busca, estas aplicações foram encontradas:

- Sofya[4] (KINNEER; DWYER; ROTHERMEL, 2006)
- EMMA[5]
- Cobertura[6]

Na Tabela 4.1 são apresentadas as principais características avaliadas na escolha da ferramenta utilizada para o cálculo de cobertura de código utilizada nesta pesquisa.

Tabela 4.1: Comparativo entre ferramentas para cálculo da cobertura de código

	Sofya	**EMMA**	**Cobertura**
Complexidade na utilização	Alta	Baixa	Baixa
Cobertura por instrução	Sim	Sim	Sim
Formato dos resultados	Console (texto)	HTML	XML

Durante a avaliação das três ferramentas, a Sofya foi descartada como alternativa para esta pesquisa por ser muito trabalhosa para ser executada, necessitando de 5 passos para gerar os dados de cobertura. A EMMA, por sua vez, mostrou-se simples na utilização com apenas dois passos necessários para o cálculo da cobertura: instrumentação e análise da cobertura. Entretanto, ela disponibiliza os resultados da análise de cobertura por instrução apenas em arquivos HTML, o que dificultaria a implementação do algoritmo de priorização.

Por fim, a Cobertura foi selecionada para ser utilizada nesta pesquisa por ser de simples utilização precisando dos mesmos dois passos da EMMA para o cálculo da cobertura (instrumentação e análise da cobertura). E também por ser a única a gerar os resultados do cálculo de cobertura por instruções em arquivos XML, o que simplificou a utilização destes dados como entrada para a priorização.

4.3 Implementação dos algoritmos de priorização

Para evitar que erros humanos ao ordenar os casos de teste com base nos grafos de chamadas e na cobertura de código prejudicassem os resultados obtidos, decidiu-se pela implementação dos algoritmos de priorização utilizados nesta pesquisa. Também com o objetivo de evitar diferenciações entre a priorização por cobertura de código e por grafo de chamadas, os algoritmos de priorização foram implementados desconsiderando-se o tipo de entrada usado.

Para tal, classes em Java foram criadas para implementar a lógica de priorização utilizada neste trabalho. A opção por utilizar esta linguagem deveu-se ao fato da ferramenta Soot

[4]http://sofya.unl.edu/
[5]http://emma.sourceforge.net/
[6]http://cobertura.github.io/cobertura/

disponibilizar o grafo de chamadas criado por ela como objetos em Java e, desta forma, seu uso facilitou a priorização usando estes grafos. O código-fonte desta implementação está disponível em um repositório *online*[7].

Também é importante mencionar que, durante a criação destas classes, o foco principal esteve na sua simplicidade e na redução da chance de erros de implementação. Por isso, um algoritmo simples porém ineficiente de ordenamento foi utilizado.

A seguir, o Algoritmo 1 apresenta em pseudo-código a implementação utilizada para o algoritmo de priorização guloso total. Seguindo a descrição apresentada na Seção 2.1.2, este algoritmo calcula a cobertura para cada um dos casos de teste independentemente e na sequência ordena os casos de teste fazendo com que os casos de teste com maior cobertura sejam executados primeiro.

Algoritmo 1 Priorização utilizando o algoritmo guloso total

Require: $CasosTesteDisponiveis$ // lista de casos de teste a serem priorizados
1: $CasosTestePriorizados \leftarrow \emptyset$ // lista dos casos de teste priorizados
2: **while** $CasosTesteDisponiveis \neq \emptyset$ **do**
3: $CasoTeste_{max} \leftarrow null$
4: **for all** $CasoTeste_i \in CasosTesteDisponiveis$ **do**
5: **if** $cobertura(CasoTeste_i) > cobertura(CasoTeste_{max})$ **then**
6: $CasoTeste_{max} \leftarrow CasoTeste_i$
7: **end if**
8: **end for**
9: $CasosTestePriorizados \leftarrow CasosTestePriorizados + CasoTeste_{max}$
10: $CasosTesteDisponiveis \leftarrow CasosTesteDisponiveis - CasoTeste_{max}$
11: **end while**
12: **return** $CasosTestePriorizados$

O Algoritmo 2 apresenta a implementação para o algoritmo guloso adicional. Seguindo novamente a descrição apresentada na Seção 2.1.2, este algoritmo diferencia-se do anterior por calcular, em cada interação, a cobertura adicional que o próximo caso de teste a ser priorizado alcança em relação ao conjunto de casos de teste já priorizados.

Com os dois algoritmos implementados ignorando-se o tipo de caso de teste sendo priorizado, a diferenciação entre o comportamento para casos de teste por cobertura de código ou por grafos de chamadas foi feita apenas na função de cálculo da cobertura. Desta forma, a função de cobertura retorna o número de instruções cobertas quando o caso de teste utiliza cobertura de código por instruções e retorna o número de nós no grafo de chamadas quando o caso de teste utiliza grafos de chamadas.

4.4 Considerações finais

Neste capítulo foi apresentada a abordagem proposta por este trabalho para a priorização de casos de teste de sistema automatizados com base nos seus grafos de chamadas. Também foi descrito o cenário em que a abordagem se propõem a ser utilizada, a motivação para escolha do critério utilizado para a priorização e os comportamentos esperados com os algoritmos estudados. Além da descrição dos passos necessários para executar a abordagem proposta. Foram

[7]https://github.com/jmeros/ppgca-thesis/tree/master/prioritization-project

Algoritmo 2 Priorização utilizando o algoritmo guloso adicional

Require: $CasosTesteDisponiveis$ // lista de casos de teste a serem priorizados
1: $CasosTestePriorizados \leftarrow \emptyset$ // lista dos casos de teste priorizados
2: $CoberturaAtual \leftarrow 0$ // cobertura atual dos casos de teste priorizados
3: **while** $CasosTesteDisponiveis \neq \emptyset$ **do**
4: $CasoTeste_{max} \leftarrow null$
5: $CoberturaAdicional_{max} = 0$ // guarda a cobertura adicional obtida pelo maior caso de teste
6: **for all** $CasoTeste_i \in CasosTesteDisponiveis$ **do**
7: $CoberturaAdicional_i \leftarrow cobertura(CoberturaAtual + CasoTeste_i)$
8: **if** $CoberturaAdicional_i > CoberturaAdicional_{max}$ **then**
9: $CasoTeste_{max} \leftarrow CasoTeste_i$
10: **end if**
11: **end for**
12: **if** $CoberturaAdicional_{max} = 0$ **then** // se nenhum caso de teste disponível melhorou a cobertura
13: $CoberturaAtual \leftarrow 0$ // limpa a cobertura atual
14: **else**
15: $CasosTestePriorizados \leftarrow CasosTestePriorizados + CasoTeste_{max}$
16: $CasosTesteDisponiveis \leftarrow CasosTesteDisponiveis - CasoTeste_{max}$
17: $CoberturaAtual \leftarrow CoberturaAdicional_{max}$
18: **end if**
19: **end while**
20: **return** $CasosTestePriorizados$

descritas também as ferramentas utilizadas e os algoritmos implementados nesta pesquisa. No próximo capítulo são apresentados os resultados obtidos com os estudos de caso executados, além de uma discussão sobre as suas implicações.

Capítulo 5

Estudos de Caso

Para bem avaliar a proposta deste trabalho, dois estudos de caso foram executados. No primeiro estudo, que é apresentado na Seção 5.1, utilizou-se uma aplicação já conhecida em estudos sobre priorização. Enquanto que no segundo, que é apresentado na Seção 5.2, optou-se pela utilização de uma aplicação desenvolvida em ambiente comercial.

5.1 Primeiro estudo de caso

Para execução do primeiro estudo de caso, as atividades definidas no método de pesquisa foram executadas da forma como estão descritas na Seção 3.2.1.

5.1.1 Busca por um projeto

Para encontrar um projeto que pudesse ser usado neste estudo de caso, todos os projetos disponibilizados no SIR foram verificados com base nos critérios definidos no método de pesquisa proposto. Infelizmente, nenhum projeto com todas estas características foi encontrado no repositório de projetos sendo utilizado.

Apesar de todos os projetos no SIR disponibilizarem o código-fonte da aplicação e uma lista de falhas semeadas, nenhum deles tinha consigo um conjunto de casos de teste de sistema automatizados. Por isso, optou-se por selecionar um projeto que permitisse que estes testes fossem escritos num curto espaço de tempo.

Após esta alteração nos critérios de seleção do projeto, o projeto escolhido foi o JMeter[1]. Por possuir uma interface gráfica simples, formada apenas por um menu principal e um painel de configuração, a sua escolha possibilitou que casos de teste automatizados fossem criados durante esta pesquisa.

5.1.2 Preparação do projeto

Para a preparação do projeto para execução deste estudo de caso, o primeiro passo a ser executado foi a escrita dos casos de teste de sistema. Entretanto, como a aplicação escolhida possui uma grande quantidade de funcionalidades disponíveis, escrever testes de sistema para testá-la completamente não seria possível dentro do tempo disponível para execução deste

[1] http://jmeter.apache.org/

trabalho. Por isso, um subconjunto de funcionalidades da aplicação foi selecionado para que os casos de teste fossem escritos.

Para definir qual subconjunto de funcionalidades seria testado, a estratégia utilizada foi a de testar todas as funções disponíveis no menu principal da aplicação. Desta forma, toda as opções disponibilizadas por meio do menu principal da aplicação foram incluídas na escrita dos casos de teste. Na Figura 5.1 pode-se visualizar este menu com suas opções (*File*, *Edit*, *Run*, *Options* e *Help*).

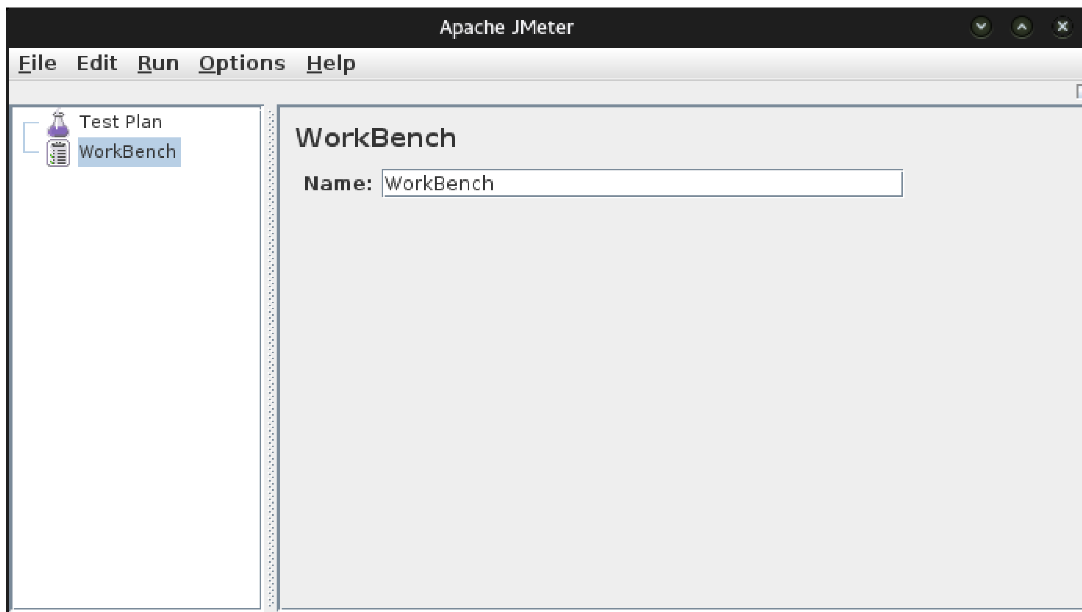


Figura 5.1: Aplicação testada no primeiro estudo de caso

Com a estratégia para criação dos casos de teste de sistema definida, iniciou-se a escrita dos casos de teste. Ao final deste processo, 97 casos de teste foram escritos para alcançar o objetivo proposto. Nas Tabelas 5.1, 5.2 e 5.3 são exemplificados alguns casos de teste escritos nesta fase do projeto. A listagem completa de todos os casos de teste escritos para este estudo de caso foi disponibilizada em um repositório *online*[2].

Tabela 5.1: Caso de teste para criar um novo arquivo usando opção do menu *File*

Referência: 1.1.	
Descrição: Criar novo arquivo usando opção de menu *File->New*.	
Pré-condições: Aplicação inicializada em estado ocioso.	
Pós-condições: Novo arquivo de teste vazio aberto na aplicação.	
Ação	**Resultado esperado**
Alterar o nome do *Workbench* atual para *New Workbench*.	Nome do *Workbench* é alterado com sucesso.
Clicar no menu principal na opção *File*.	As opções do menu *File* são mostradas.
Clicar na opção *New*.	As alterações atuais são descartadas e um novo arquivo de teste vazio é aberto.

[2]https://github.com/jmeros/ppgca-thesis/tree/master/PrimeiroEstudo_CasosTeste

Tabela 5.2: Caso de teste para abrir um arquivo usando opção do menu *File*

<table>
<tr><td colspan="2">Referência: 1.4.</td></tr>
<tr><td colspan="2">Descrição: Abrir arquivo usando opção de menu File->Open.</td></tr>
<tr><td colspan="2">Pré-condições: Aplicação inicializada em estado ocioso.</td></tr>
<tr><td colspan="2">Pós-condições: Conteúdo do arquivo de teste aberto é mostrado na aplicação.</td></tr>
<tr><th>Ação</th><th>Resultado esperado</th></tr>
<tr><td>Clicar no menu principal na opção File.</td><td>As opções do menu File são mostradas.</td></tr>
<tr><td>Clicar na opção Open</td><td>Janela para seleção do arquivo a ser aberto é mostrada.</td></tr>
<tr><td>Selecionar o arquivo que será aberto.</td><td>Nome do arquivo selecionado é mostrado na caixa de texto.</td></tr>
<tr><td>Clicar na botão Open na janela de seleção.</td><td>Conteúdo do arquivo selecionado é mostrado na aplicação.</td></tr>
</table>

Tabela 5.3: Caso de teste para adicionar um *Thread Group* ao *Test Plan*.

<table>
<tr><td colspan="2">Referência: 2.1.1.</td></tr>
<tr><td colspan="2">Descrição: Adicionar um Thread Group usando a opção menu Edit->Add->Thread Group.</td></tr>
<tr><td colspan="2">Pré-condições: Aplicação inicializada em estado ocioso.</td></tr>
<tr><td colspan="2">Pós-condições: Elemento Thread Group é adicionado ao Test Plan.</td></tr>
<tr><th>Ação</th><th>Resultado esperado</th></tr>
<tr><td>Clicar no elemento Test Plan do teste atual aberto na aplicação.</td><td>O elemento Test Plan é selecionado.</td></tr>
<tr><td>Clicar no menu principal na opção Edit.</td><td>As opções do menu Edit são mostradas.</td></tr>
<tr><td>Navegar até a opção Add</td><td>As sub-opções do menu Add são mostradas.</td></tr>
<tr><td>Clicar na sub-opção Thread Group.</td><td>Um elemento do tipo Thread Group é adicionado ao Test Plan.</td></tr>
</table>

A partir destes casos de teste de sistema foram então criados os casos de teste automatizados. Como a aplicação sendo testada utiliza componentes Java/Swing para montagem da sua interface gráfica, optou-se pela utilização das bibliotecas JUnit[3] e UISpec4J[4] para criação dos casos de teste.

A biblioteca JUnit foi escolhida por ser comumente utilizada na escrita de casos de teste automatizados na linguagem Java. Por sua vez, a biblioteca UISpec4J foi escolhida por permitir a simulação da iteração do usuário com a interface gráfica sem acesso ao código da aplicação sendo testada. Todavia, com o uso da ferramenta Averroes durante a criação dos grafos de chamadas, os grafos utilizados neste estudo acabam por ignorar as bibliotecas sendo utilizadas e pode-se afirmar que a escolha destas bibliotecas não tem grande impacto sobre os resultados obtidos.

Para testar os 97 casos de teste, 5 classes de teste foram criadas (uma classe para cada menu da aplicação). E dentro dessas classes, um método de teste para cada um dos casos de teste escritos. Seguindo os conceitos da programação orientada a objetos, buscou-se sempre que

[3]http://junit.org/
[4]http://www.uispec4j.org/

possível a utilização de abstrações e do encapsulamento de funções. O código-fonte completo para todos os casos de teste está disponível a comunidade em um repositório *online*[5].

Para mostrar como esses casos de teste foram automatizados, o Código 1 apresenta o caso de teste automatizado referente ao caso de teste descrito na Tabela 5.1. Da mesma forma, o Código 2 apresenta o caso de teste descrito na Tabela 5.2 e o Código 3 apresenta o caso de teste descrito na Tabela 5.3.

Código 1 Caso de teste automatizado para testar opção de menu *File->New*

```
1  /**
2  * Test case that changes something on the current project (rename WorkBench)
3  * and then start a new project by selection File->New on the menu.
4  *
5  * Test reference: 1.1.
6  */
7  @Test
8  public void newFileUsingMenu() {
9        // Select workbench
10       mainWindow.getTree()
11             .selectWorkbench()
12
13       // Change workbench name on text box
14             .changeWorkbenchName("New WorkBench");
15
16       // Verify workbench name was changed
17       mainWindow.getTree().assertEquals(
18             "Root\n" +
19             "  Test Plan\n" +
20             "  New WorkBench");
21
22       // Select new option on File menu
23       mainWindow.getMenu().newFileUsingFileMenu(false);
24
25       // Verify workbench name was returned to default value
26       mainWindow.getTree().assertEquals(
27             "Root\n" +
28             "  Test Plan\n" +
29             "  WorkBench");
30 }
```

Por fim, para finalizar a preparação do projeto, selecionou-se uma das versões da aplicação para ser usada neste estudo de caso. Como o projeto disponibilizado no SIR possui 5 diferentes versões da aplicação, cada uma delas contento de 13 a 20 falhas semeadas manualmente, um critério foi definido para seleção da versão a ser usada neste estudo.

O critério utilizado para esta seleção foi o da escolha da versão com o maior número de falhas detectáveis a partir do menu principal da aplicação. Assim sendo, a versão `v1_8` foi escohida por possuir 5 falhas detectáveis a partir do menu principal da aplicação num total de 19 falhas semeadas.

Na Tabela 5.4 são apresentadas cada uma destas falhas. Além disso, são indicadas também quais delas foram detectadas pelos casos de teste criados para este estudo de caso.

[5]https://github.com/jmeros/ppgca-thesis/tree/master/ApacheJMeter_uispec4j-test

Código 2 Caso de teste automatizado para abrir um arquivo usando a opção de menu *File->Open*

```
/**
* Test case open a template testfile using the File menu
* and verify if the testfile was loaded sucessfully.
*
* Test reference: 1.4.
*/
@Test
public void openFileUsingMenu() {
      // Open proxy test plan (testfiles/proxy.jmx)
      String proxyJmx =
            new File(testfilesPath, "proxy.jmx").getAbsolutePath();
      mainWindow.getMenu().openFileUsingFileMenu(proxyJmx);

      // Check tree view was populated with proxy test plan
      mainWindow.getTree().assertEquals(
            "Root\n"     +
            "  Test Plan\n" +
            "    Thread Group\n" +
            "      Simple Controller\n" +
            "        HTTP Request Defaults\n" +
            "  WorkBench");

      mainWindow.getMenu().newFileUsingAccelerator(false);
}
```

Código 3 Caso de teste automatizado para adicionar um ThreadGroup usando a opção de menu *Edit->Add->Thread Group*

```
/**
* Add a Thread Group element to Test Plan using the Add
* submenu inside Edit menu. Verify if it was added successfully
* and recover initial state of the application (new file).
*
* Test reference: 2.1.1.
*/
@Test
public void addThreadGroupToTestPlan() {
      // Select Test Plan element on treeview
      mainWindow.getTree().selectTestPlan();

      // Add new element using Edit menu
      mainWindow.getMenu().addThreadGroupFromMenu();

      // Check treeview to check new element was inserted
      // and is selected
      checkElementAddedToTestPlan("Thread Group");

      // Clear application using new file
      mainWindow.getMenu().newFileUsingFileMenu(true);
}
```

Tabela 5.4: Falhas semeadas na versão escolhida para o primeiro estudo de caso

Identificador no projeto	**Descrição da falha**	**Detectável**
LD_HD_1	Exceção inesperada ocorre ao carregar arquivo inválido.	Sim
SV_HD_1	Arquivo salvo a partir do menu *File->Save* fica com o seu conteúdo vazio.	Sim
JMB_HD_1	O mnemônico do menu para a opção de salvar arquivo não funciona.	Sim
MF_HD_1	Arquivo salvo a partir do menu *Edit->Save* fica com o seu conteúdo vazio.	Sim
SS_HD_1	Serialização do tag <string> não é feita corretamente para arquivo.	Sim
FDG_AK_1	Caminho incorreto do último diretório visitado ao abrir arquivo é salvo.	Não
FD_HD_1	Alteração no tipo de objeto passado ao seletor de arquivo na sua instanciação.	Não
MF_AK_1	Altera tipo da exceção capturada ao ocorrer falha na montagem do menu.	Não
PTM_HD_1	Ao adicionar nova coluna a tabela, dados da coluna são gerados incorretamente.	Não
PTM_HD_2	Ao remover uma coluna da tabela, a interface gráfica não é atualizada.	Não
JMU_AK_1	Propriedade JMETER_HOME é lida incorretamente do arquivo de propriedades.	Não
URM_AK_1	Componente que modifica URL usando expressões regulares não funciona corretamente.	Não
HTS_AK_1	Componente *HTTPSampler* não consegue estabelecer conexão HTTP.	Não
PW_AK_1	Cabeçalhos gerados pelo componente *PostWriter* são configurados incorretamente.	Não
IC_HD_1	Falha interna no componente *InterleaveControl*.	Não
IC_HD_2	Falha interna no componente *InterleaveControl*.	Não
IC_HD_3	Falha interna no componente *InterleaveControl*.	Não
ILC_AK_1	Falha interna no componente *InterleaveControl*.	Não
CF_HD_1	Falha interna no componente *CompoundFunction*.	Não

5.1.3 Coleta dos dados

Para que fossem coletados os resultados para cada um dos casos de teste e também fosse obtida a ordem original de execução dos casos de teste, primeiramente o conjunto de casos de teste automatizados foi executado. Desta forma, verificou-se quais casos de teste foram capazes de detectar falhas na aplicação e quais foram executados com sucesso.

Esta execução também foi utilizada para verificar o tempo necessário para a execução do conjunto de casos de teste. Neste estudo o tempo total de execução do conjunto de 97 casos de teste foi de aproximadamente 10 minutos.

Por sua vez, para que fossem geradas as 10 ordens randômicas de execução do conjunto de casos de teste, utilizou-se o método *shuffle* da classe *Collections* de Java para randomizar o conjunto de casos de teste. A ordem original de execução e o resultado da criação destas ordens randômicas foram disponibilizados em um repositório *online*[6].

Na Tabela 5.5 é apresentada a ordem original de execução dos casos de teste neste estudo de caso. Seguindo o comportamento padrão da ferramenta sendo utilizada para esta execução, cada classe de teste tem todos os seus métodos de teste executados antes que a execução da próxima classe tenha início.

Tabela 5.5: Ordem de execução original dos casos de teste no primeiro estudo de caso

Posição	Classe de teste	Método de teste
1	RunMenuTest	Todos os métodos
2	EditMenuTest	Todos os métodos
3	FileMenuTest	Todos os métodos
4	OptionsMenuTest	Todos os métodos
5	HelpMenuTest	Todos os métodos

Para possibilitar a coleta dos dados de cobertura de código da aplicação sendo testada, procedeu-se com a instrumentação da aplicação usando a ferramenta Cobertura. A seguir, cada caso de teste foi executado individualmente e as informações sobre a sua cobertura alcançada foram obtidas. Com as informações sobre a cobertura de código na aplicação sendo testada para cada caso de teste coletadas, a ferramenta Cobertura foi executada novamente para que os dados de cobertura de código por instrução para cada caso de teste fossem exportados para arquivos XML.

Por fim, para coletar os dados necessários para a criação dos grafos de chamadas, utilizou-se a ferramenta Averroes para remover as dependências dos casos de teste para as bibliotecas sendo utilizadas e preparar as classes para a criação dos grafos. Este processo gerou um arquivo JAR contendo as classes com os casos de teste automatizados e outro arquivo JAR com as classes referentes as bibliotecas sendo utilizadas porém com funções vazias.

5.1.4 Aplicação das abordagens escolhidas

Para aplicar a abordagem proposta, os arquivos gerados pela ferramenta Averroes foram utilizados para criação dos grafos de chamadas para cada caso de teste utilizando a ferramenta Soot. Estes grafos de chamadas foram então utilizados com os algoritmos apresentados na Seção 4.3.

[6]https://github.com/jmeros/ppgca-thesis/tree/master/PrimeiroEstudo_OrdensExecucao

Para ilustrar os grafos de chamadas gerados, a Figura 5.2 apresenta o grafo criado para o caso de teste automatizado referente ao Código 1. Da mesma forma, a Figura 5.3 apresenta o grafo criado para o Código 2 e a Figura 5.4 apresenta o grafo criado para o Código 3.

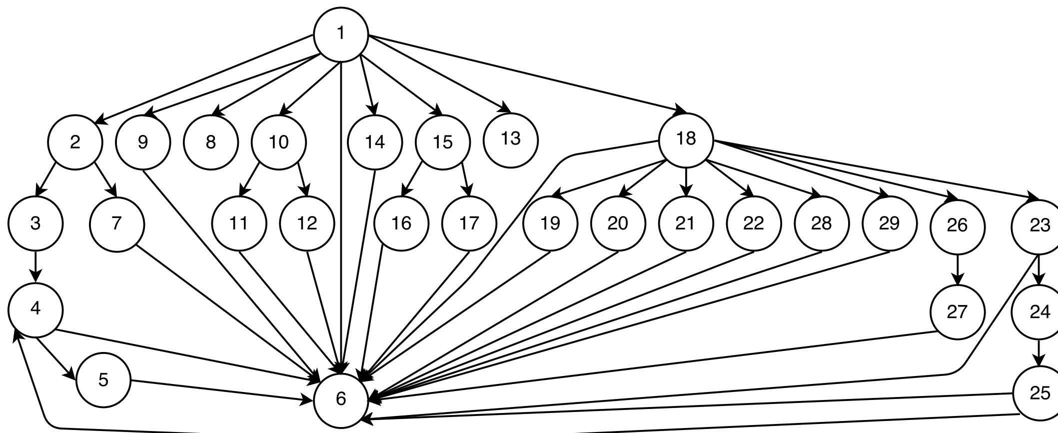
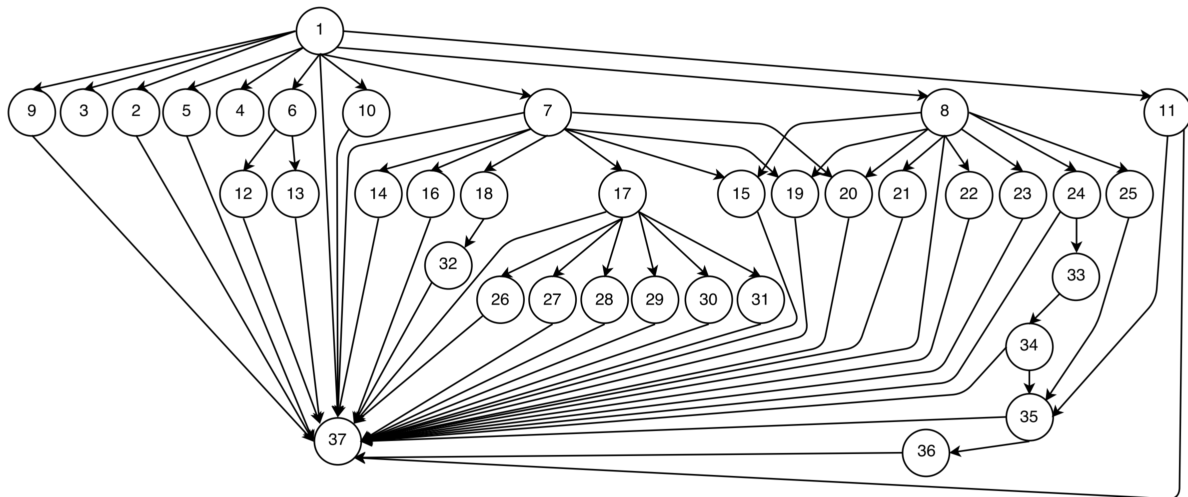


Figura 5.2: Grafo gerado para o caso de teste da opção de menu *File->New*

Como este trabalho se propôs a verificar o uso de dois algoritmos de priorização com a abordagem proposta, os grafos de chamadas gerados pela ferramenta Soot foram priorizados duas vezes. Na primeira execução utilizando-se o algoritmo guloso total e na segunda execução o algoritmo guloso adicional. O resultado da priorização com estes dois algoritmos pode ser visto em arquivos contendo as ordens de execução geradas que estão disponíveis em um repositório *online*[7].

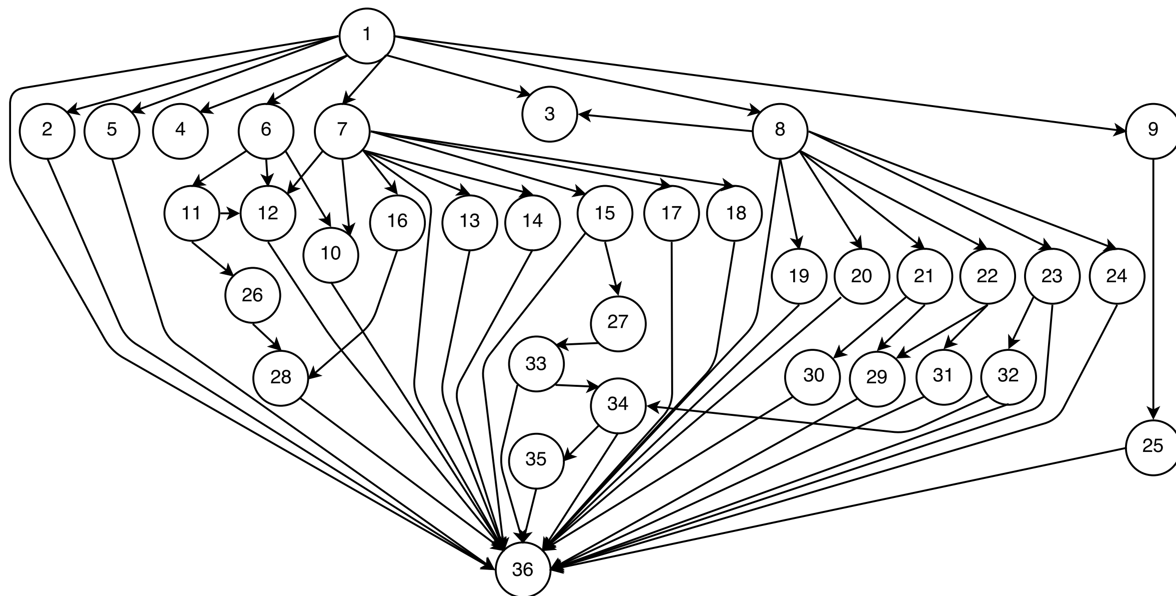
Para exemplificar como as ordens de execução para a abordagem proposta foram criadas, a Tabela 5.6 mostra os primeiros 10 casos de teste priorizados com o algoritmo guloso

[7]https://github.com/jmeros/ppgca-thesis/tree/master/PrimeiroEstudo_OrdensExecucao



Legenda:
1 - org.apache.jmeter.gui.test.FileMenuTest.openFileUsingMenu()
2 - java.lang.Boolean.booleanValue()
3 - org.apache.jmeter.uispec4j.JMeterMainWindow.getTree()
4 - org.apache.jmeter.uispec4j.JMeterMainWindow.getMenu()
5 - java.lang.Boolean.<clinit>()
6 - org.apache.jmeter.uispec4j.JMeterTree.assertEquals(java.lang.String)
7 - org.apache.jmeter.uispec4j.JMeterMenu.openFileUsingFileMenu(java.lang.String)
8 - org.apache.jmeter.uispec4j.JMeterMenu.newFileUsingAccelerator(boolean)
9 - java.io.File.<clinit>()
10 - java.io.File.getAbsolutePath()
11 - java.io.File.<init>(java.lang.String, java.lang.String)
12 - org.uispec4j.assertion.Assertion__Averroes.check()
13 - org.uispec4j.Tree.contentEquals(java.lang.String)
14 - org.uispec4j.MenuItem.triggerClick()
15 - org.uispec4j.interception.WindowInterceptor.process(org.uispec4j.interception.WindowHandler)
16 - org.uispec4j.MenuItem.getSubMenu(java.lang.String)
17 - org.apache.jmeter.uispec4j.JMeterMenu.openFilenameOnFileChooser(java.lang.String)
18 - org.apache.jmeter.uispec4j.JMeterMenu.accessFileMenu()
19 - org.uispec4j.interception.WindowInterceptor.run()
20 - org.uispec4j.interception.WindowInterceptor.init(org.uispec4j.Trigger)
21 - org.uispec4j.Key.<clinit>()
22 - org.uispec4j.Key.control(org.uispec4j.Key)
23 - org.uispec4j.AbstractUIComponent.typeKey(org.uispec4j.Key)
24 - org.apache.jmeter.uispec4j.JMeterMenu.closeDialogWithoutSave()
25 - org.apache.jmeter.uispec4j.JMeterMenu$1.<init>(org.apache.jmeter.uispec4j.JMeterMenu)
26 - org.uispec4j.interception.FileChooserHandler.assertAcceptsFilesOnly()
27 - org.uispec4j.interception.FileChooserHandler.<clinit>()
28 - org.uispec4j.interception.FileChooserHandler.titleEquals(java.lang.String)
29 - org.uispec4j.interception.FileChooserHandler.init()
30 - org.uispec4j.interception.FileChooserHandler.assertIsOpenDialog()
31 - org.uispec4j.interception.FileChooserHandler.select(java.lang.String)
32 - org.uispec4j.MenuBar.getMenu(java.lang.String)
33 - org.apache.jmeter.uispec4j.JMeterMenu$10.<init>(org.apache.jmeter.uispec4j.JMeterMenu)
34 - org.uispec4j.interception.WindowHandler.<init>()
35 - java.lang.Object.<init>()
36 - java.lang.Object.finalize()
37 - java.lang.Object.<clinit>()

Figura 5.3: Grafo gerado para o caso de teste que abre um arquivo usando a opção de menu *File->Open*



Legenda:
1 - org.apache.jmeter.gui.test.EditMenuTest.addThreadGroupToTestPlan()
2 - java.lang.Boolean.booleanValue()
3 - org.apache.jmeter.uispec4j.JMeterMainWindow.getTree()
4 - org.apache.jmeter.uispec4j.JMeterMainWindow.getMenu()
5 - java.lang.Boolean.<clinit>()
6 - org.apache.jmeter.uispec4j.JMeterMenu.addThreadGroupFromMenu()
7 - org.apache.jmeter.uispec4j.JMeterMenu.newFileUsingFileMenu(boolean)
8 - org.apache.jmeter.gui.test.EditMenuTest.checkElementAddedToTestPlan(java.lang.String)
9 - org.apache.jmeter.uispec4j.JMeterTree.selectTestPlan()
10 - org.uispec4j.MenuItem.click()
11 - org.apache.jmeter.uispec4j.JMeterMenu.accessAddSubMenu()
12 - org.uispec4j.MenuItem.getSubMenu(java.lang.String)
13 - org.uispec4j.MenuItem.triggerClick()
14 - org.uispec4j.interception.WindowInterceptor.process(org.uispec4j.interception.WindowHandler)
15 - org.apache.jmeter.uispec4j.JMeterMenu.closeDialogWithoutSave()
16 - org.apache.jmeter.uispec4j.JMeterMenu.accessFileMenu()
17 - org.uispec4j.interception.WindowInterceptor.run()
18 - org.uispec4j.interception.WindowInterceptor.init(org.uispec4j.Trigger)
19 - java.lang.AbstractStringBuilder.<clinit>()
20 - java.lang.StringBuilder.append(java.lang.String)
21 - org.apache.jmeter.uispec4j.JMeterTree.assertEquals(java.lang.String)
22 - org.apache.jmeter.uispec4j.JMeterTree.assertSelected(java.lang.String)
23 - java.lang.StringBuilder.<init>(java.lang.String)
24 - java.lang.StringBuilder.toString()
25 - org.uispec4j.Tree.select(java.lang.String)
26 - org.apache.jmeter.uispec4j.JMeterMenu.accessEditMenu()
27 - org.apache.jmeter.uispec4j.JMeterMenu$10.<init>(org.apache.jmeter.uispec4j.JMeterMenu)
28 - org.uispec4j.MenuBar.getMenu(java.lang.String)
29 - org.uispec4j.assertion.Assertion__Averroes.check()
30 - org.uispec4j.Tree.contentEquals(java.lang.String)
31 - org.uispec4j.Tree.selectionEquals(java.lang.String)
32 - java.lang.AbstractStringBuilder.<init>()
33 - org.uispec4j.interception.WindowHandler.<init>()
34 - java.lang.Object.<init>()
35 - java.lang.Object.finalize()
36 - java.lang.Object.<clinit>()

Figura 5.4: Grafo gerado para o caso de teste que adiciona um ThreadGroup usando a opção de menu *Edit->Add->Thread Group*

total. Da mesma forma, a Tabela 5.7 mostra os primeiros 10 casos de teste priorizados com o algoritmo guloso adicional.

Tabela 5.6: Ordem de execução para a abordagem proposta usando guloso total no primeiro estudo de caso

Posição	Classe de teste	Método de teste
1	EditMenuTest	saveFileFromWorkBench
2	EditMenuTest	saveFileFromTestPlan
3	FileMenuTest	saveFileAsUsingMnemonic
4	FileMenuTest	saveFileUsingMnemonic
5	FileMenuTest	openFileUsingMnemonic
6	EditMenuTest	openFileOnWorkBench
7	EditMenuTest	cutThreadGroupFromTestPlan
8	EditMenuTest	copyThreadGroupFromTestPlan
9	EditMenuTest	openFileOnTestPlan
10	FileMenuTest	saveFileAsUsingAccelerator

Tabela 5.7: Ordem de execução para a abordagem proposta usando guloso adicional no primeiro estudo de caso

Posição	Classe de teste	Método de teste
1	EditMenuTest	saveFileFromWorkBench
2	OptionsMenuTest	testThreadNumFunctionHelper
3	FileMenuTest	saveFileAsUsingMnemonic
4	RunMenuTest	stopTestExecutionUsingKeyboard
5	EditMenuTest	cutThreadGroupFromTestPlan
6	HelpMenuTest	accessHelpFromKeyboard
7	RunMenuTest	startRemoteTestExecutionUsingMenu
8	EditMenuTest	disableDisabledElement
9	FileMenuTest	newFileUsingMnemonic
10	HelpMenuTest	accessAboutFromMenu

Em ambos os casos, o tempo necessário para gerar os grafos de chamadas para os 97 casos de teste foi de aproximadamente 30 segundos. Entretanto, o tempo para priorizar os casos de teste foi de 8 milissegundos para o algoritmo guloso total e de 34 milissegundos para o algoritmo guloso adicional.

Da mesma forma, para obter os resultados com a abordagem tradicional utilizando-se os dois algoritmos de priorização, os arquivos XML obtidos da ferramenta Cobertura foram utilizados com os algoritmos apresentados na Seção 4.3.

Exemplificando também as ordens de execução para a abordagem utilizando a cobertura de código por instrução, a Tabela 5.8 apresenta os 10 primeiros casos de teste priorizados pelo algoritmo guloso total. E a Tabela 5.9 apresenta os 10 primeiros casos de teste priorizados pelo algoritmo guloso adicional.

Tabela 5.8: Ordem de execução para a abordagem tradicional usando guloso total no primeiro estudo de caso

Posição	Classe de teste	Método de teste
1	RunMenuTest	stopTestExecutionUsingMenu
2	RunMenuTest	startTestExecutionUsingKeyboard
3	RunMenuTest	startTestExecutionUsingMenu
4	EditMenuTest	openFileOnWorkBench
5	EditMenuTest	openFileOnTestPlan
6	FileMenuTest	saveFileAsUsingMenu
7	FileMenuTest	saveFileAsUsingAccelerator
8	FileMenuTest	saveFileAsUsingMnemonic
9	FileMenuTest	openFileUsingMenu
10	FileMenuTest	openFileUsingMnemonic

Tabela 5.9: Ordem de execução para a abordagem tradicional usando guloso adicional no primeiro estudo de caso

Posição	Classe de teste	Método de teste
1	RunMenuTest	stopTestExecutionUsingMenu
2	EditMenuTest	openFileOnWorkBench
3	FileMenuTest	openInvalidFileUsingMenu
4	FileMenuTest	saveFileAsUsingMenu
5	EditMenuTest	enableDisabledElement
6	OptionsMenuTest	testCounterFunctionHelper
7	EditMenuTest	saveFileFromWorkBench
8	HelpMenuTest	accessAboutFromMenu
9	EditMenuTest	removeThreadGroupFromTestPlan
10	EditMenuTest	addSplineVisualizerToTestPlan

5.1.5 Avaliação dos resultados obtidos

A partir das ordens de execução para cada uma das abordagens sendo avaliadas e das informações a respeito da capacidade de detecção de falhas de cada caso de teste, utilizou-se a fórmula de cálculo da APFD para avaliar cada uma das abordagens. No caso das ordens randômicas, o valor relativo a APFD de cada execução foi calculado e então a média aritmética das 10 execuções foi considerada como resultado final.

Na Figura 5.5 pode-se visualizar os resultados utilizando a métrica APFD obtidos com cada uma das abordagens aplicadas neste estudo de caso.

Após aplicar a fórmula da APFD para cada uma das abordagens, as execuções não ordenadas dos casos de teste obtiveram os menores resultados para a APFD. Tendo a ordem original de execução obtido uma APFD de 29,59% e a média das ordens randômicas de execução 59,98%.

No caso das ordens de execução utilizando a abordagem proposta, a execução gerada a partir do algoritmo guloso total conseguiu superar apenas as execuções não ordenadas com uma APFD de 67,94%. Entretanto, a execução gerada a partir do algoritmo guloso adicional obteve o melhor resultado entre todas as abordagens chegando a 91,03%.

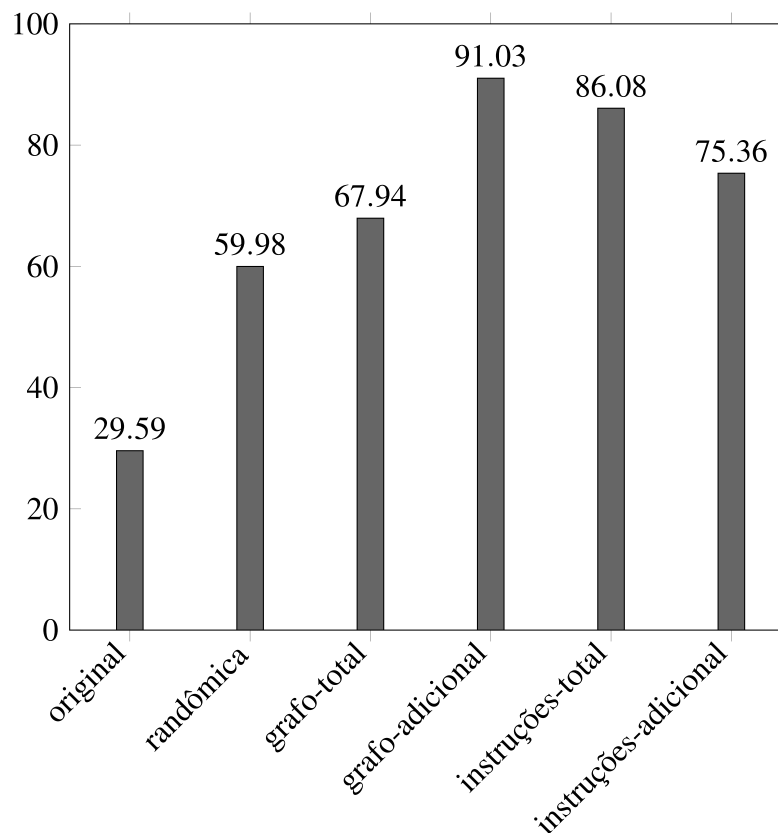


Figura 5.5: APFD para as ordens de execução avaliadas no primeiro estudo de caso

Já para as ordens de execução utilizando a abordagem tradicional de cobertura de código, a execução gerada a partir do guloso total obteve resultado superior ao guloso adicional com valores para a APFD de 86,08% e 75,36%, respectivamente. Estes resultados foram melhores que as execuções não ordenadas dos casos de teste e que a abordagem proposta utilizando o algoritmo guloso total.

5.2 Segundo estudo de caso

O projeto utilizado neste estudo de caso foi desenvolvido em uma instituição pública federal e permite que o acompanhamento de sessões de julgamento seja feito eletronicamente por meio de uma página Web. Implementado em Java, esta aplicação é executada em um servidor de aplicações JBoss e obtém os dados a respeito das sessões de julgamento programadas por meio de uma conexão com um banco de dados Oracle.

Os casos de teste automatizados criados para testá-la foram escritos em Java utilizando a biblioteca Selenium[8] e testam todas as funcionalidades da aplicação. No total são 97 casos de teste, divididos em 15 classes, sendo que cada classe testa uma funcionalidade da aplicação. O código-fonte completo para todos os casos de teste está disponível a comunidade em um repositório *online*[9].

[8]http://www.seleniumhq.org/

[9]https://github.com/jmeros/ppgca-thesis/tree/master/acompspje_selenium-tests

Para demonstrar como esses casos de teste foram escritos, no Código 4 é apresentado um exemplo de caso de teste que verifica se o acesso ao sistema como magistrado funciona corretamente. No Código 5 é apresentado outro caso de teste que verifica se o voto completo é mostrado corretamente para o secretário da sessão.

Código 4 Caso de teste automatizado para testar acesso como magistrado

```
1  /**
2  * Logar sem e-Token no perfil de Magistrado e verifica que o painel de
3  * sessoes mostra apenas a opcao de a sessao como magistrado.
4  */
5  @Test
6  public void logarComoMagistrado() {
7
8        try {
9              // Alterar perfil no banco para Magistrado
10             BDUtils.getInstance()
11                   .alterarPerfilUsuario(PerfilUsuario.Magistrado);
12
13             // Entrar no sistema usando o perfil de magistrado
14             new TelaInicial(driver)
15                   .autenticarUsuario()
16
17             // Validar que a tela de sessoes do magistrado
18                   .validarPerfilMagistrado(SESSAO.getID_SESSAO_PJE());
19       }
20       finally {
21             // Retornar o perfil do usuario para Secretario
22             BDUtils.getInstance()
23                   .alterarPerfilUsuario(PerfilUsuario.Secretario);
24       }
25 }
```

Na Tabela 5.10 são apresentadas as 12 falhas detectadas por estes casos de teste durante o desenvolvimento do projeto.

5.2.1 Coleta dos dados

Os dados sobre este projeto foram coletados no ambiente de desenvolvimento da instituição e foram obtidos por meio da análise de históricos de execução e de registros de falhas durante o desenvolvimento do projeto. Para obter a ordem original de execução dos casos de teste, o histórico de execuções dos casos de teste na ferramenta de integração contínua foram avaliados. O tempo médio de execução para o conjunto completo de casos de teste deste projeto é de 1 hora e 10 minutos.

Assim como no primeiro estudo de caso, utilizando-se a listagem de todos os casos de teste executados, as 10 ordens randômicas de execução foram geradas usando o método *shuffle* da classe *Collections* de Java. E a ordem original de execução dos casos de teste, assim como o resultado da criação destas ordens randômicas estão disponíveis em um repositório *online*[10].

[10]https://github.com/jmeros/ppgca-thesis/tree/master/SegundoEstudo_OrdensExecucao

Código 5 Casos de teste automatizado para testar visualização de voto

```
/**
* Verificar que o secretario consegue visualizar o voto completo de um
* processo da sessao atual.
*/
@Test
public void visualizarVotoCompletoSecretario() {

      // Selecionar os processos a serem utilizados no teste
      RoteiroPautaSessao primeiroProcesso =
            encontrarPrimeiroProcessoDaSessao();
      RoteiroPautaSessao processoApregoado =
            encontrarPrimeiroProcessoNoEstado(EstadoProcesso.APREGOADO);

      // Entrar no sistema
      new TelaInicial(driver)
            .autenticarUsuario()

      // Acessar lista de sessoes disponiveis e escolher uma sessao
            .entrarSessaoComoSecretarioPorNumeroPauta(
                  SESSAO.getID_SESSAO_PJE(), PROCESSOS_PAUTA.size())
            .validarProcessoSelecionado(processoApregoado)

      // Seleciona o primeiro processo da lista na sessao
            .selecionarProcesso(primeiroProcesso)

      // Validar os detalhes do primeiro processo (todos os dados completos)
            .validarProcessoSelecionado(primeiroProcesso)

      // Selecionar a opcao para visualizar o voto completo
            .visualizarVotoCompleto()

      // Validar o conteudo do voto completo
            .validarVotoCompleto(primeiroProcesso);
}
```

Tabela 5.10: Falhas detectadas pelos casos de teste no segundo estudo de caso

Identificador no projeto	Descrição da falha
ACOMPSPJE-112	Quadro-resumo sem espaços entre os estados dos processos.
ACOMPSPJE-188	Caracteres especiais no editor de dispositivo não são salvos.
ACOMPSPJE-197	Não é possível sair do sistema (botão Sair não existe).
ACOMPSPJE-206	Data e hora do cabeçalho do relatório estão formatadas incorretamente.
ACOMPSPJE-214	Magistrado recebe mensagem de processo apregoado ao secretário mover seleção do processo apregoado.
ACOMPSPJE-282	Mensagem é mostrada em inglês quando o filtro é aplicado com número de processo inexistente.
ACOMPSPJE-299	Magistrado tem acesso ao menu de configurações.
ACOMPSPJE-303	NullPointerException quando usuário não possui um perfil associado.
ACOMPSPJE-304	Gabinete tem acesso ao menu de configurações.
ACOMPSPJE-336	Vírgula posicionada incorretamente na mensagem apresentada na vista regimental.
ACOMPSPJE-346	Usuário com perfil SECRETARIO_GABINETE não encerra a sessão.
ACOMPSPJE-348	Classe judicial do processo não é mostrada corretamente nos detalhes do processo selecionado.

Na Tabela 5.11 é apresentada a ordem original de execução dos casos de teste neste estudo de caso. Seguindo o comportamento padrão do servidor de integração contínua para a sua execução, cada classe de teste tem todos os seus métodos de teste executados antes que a execução da próxima classe tenha início.

Tabela 5.11: Ordem de execução original dos casos de teste no segundo estudo de caso

Posição	Classe de teste	Método de teste
1	TC10_EncerrarSessaoTest	Todos os métodos
2	TC03_PainelAcompanhamentoTest	Todos os métodos
3	TC02_ManterAtualizacaoAutomaticaTest	Todos os métodos
4	TC13_GerarRoteiroSessaoTest	Todos os métodos
5	TC12_GerarRelatorioSessaoTest	Todos os métodos
6	TC07_VisualizarVotoCompletoTest	Todos os métodos
7	TC11_ManterCadastroSustentacaoOralTest	Todos os métodos
8	TC06_EditarDispositivoTest	Todos os métodos
9	TC05_ApregoarProcessoTest	Todos os métodos
10	TC09a_AlterarStatusProcessoTest	Todos os métodos
11	TC09b_AlterarStatusProcessoTest	Todos os métodos
12	TC01_AcessarSistemaTest	Todos os métodos
13	TC04_MarcarProcessoComoJulgadoTest	Todos os métodos
14	TC15_ManterCadastroVisibilidadeProcuradoresTest	Todos os métodos
15	TC08_LocalizarProcessoNaSessaoTest	Todos os métodos

Para obter as informações relativas ao histórico de falhas detectadas durante o projeto de desenvolvimento desta aplicação e dos casos de teste que as detectaram, a ferramenta utili-

zada pela instituição para rastreio das falhas foi consultada. Por fim, o código-fonte dos casos de teste de sistema automatizados foi obtido no repositório do projeto dentro do servidor de controle de versão.

Seguindo a mesma estratégia adotada no primeiro estudo, a ferramenta Averroes foi utilizada para remover as dependências dos casos de teste para as bibliotecas sendo utilizadas e preparar as classes para a criação dos grafos de chamadas. Somente após esta preparação inicial que o código-fonte foi usado para aplicação da abordagem proposta.

5.2.2 Aplicação da abordagem proposta

Utilizando-se a ferramenta Soot juntamente aos arquivos gerados pela ferramenta Averroes, os grafos de chamadas para cada caso de teste foram criados. E estes grafos de chamadas foram então priorizados utilizando-se os algoritmos apresentados na Seção 4.3.

Para exemplificar a complexidade dos grafos gerados para este estudo de caso, o grafo gerado para o caso de teste apresentado no Código 4 continha 96 nós. Já o grafo gerado para o caso de teste apresentado no Código 5 continha 199 nós. Estes grafos não são apresentados no documento devido a quantidade de nós gerados.

Da mesma forma que no primeiro estudo, essa atividade foi repetida por duas vezes para que os casos de teste fossem priorizados com os dois algoritmos sendo avaliados: guloso total e guloso adicional. O resultado da priorização com estes dois algoritmos pode ser visto em arquivos contendo as ordens de execução geradas que estão disponíveis em um repositório *online*[10].

Para exemplificar como as ordens de execução para a abordagem proposta foram criadas, a Tabela 5.12 mostra os primeiros 10 casos de teste priorizados com o algoritmo guloso total. Da mesma forma, a Tabela 5.13 mostra os primeiros 10 casos de teste priorizados com o algoritmo guloso adicional.

Tabela 5.12: Ordem de execução para a abordagem proposta usando guloso total no segundo estudo de caso

Posição	**Classe de teste**	**Método de teste**
1	TC09a_AlterarStatusProcessoTest	vistaRegimentalProcessoNãoJulgado
2	TC09a_AlterarStatusProcessoTest	marcarPreferencialProcesso
3	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoSemNenhumNao-JulgadoDisponivel
4	TC10_EncerrarSessaoTest	encerrarSessaoDisponivelSecretário
5	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoSemNenhumNao-JulgadoOuVistaMesaDisponivel
6	TC05_ApregoarProcessoTest	apregoarProcessoNãoJulgado
7	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoRetirado
8	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoVistaRegimental
9	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoRevisar
10	TC04_MarcarProcessoComoJulgadoTest	julgarProcessoNãoJulgado

Neste estudo de caso, o tempo necessário para gerar os grafos de chamadas para todos os casos de teste foi de aproximadamente 35 segundos. Por sua vez, o tempo necessário para

Tabela 5.13: Ordem de execução para a abordagem proposta usando guloso adicional no segundo estudo de caso

Posição	Classe de teste	Método de teste
1	TC09a_AlterarStatusProcessoTest	vistaRegimentalProcessoNãoJulgado
2	TC12_GerarRelatorioSessaoTest	gerarRelatorioDaSessaoEncerrada
3	TC03_PainelAcompanhamentoTest	verificarListaProcessosSecretario
4	TC11_ManterCadastroSustentacao-OralTest	alterarInscricaoPosicaoJaUtilizada-SustentacaoOral
5	TC07_VisualizarVotoCompletoTest	visualizarVotoCompletoProcurador
6	TC13_GerarRoteiroSessaoTest	gerarRoteiroSessaoDisponivel-SecretarioGabinete
7	TC15_ManterCadastroVisibilidade-ProcuradoresTest	ativarVisibilidadeProcuradores
8	TC10_EncerrarSessaoTest	encerrarSessaoDisponivelSecretário
9	TC09a_AlterarStatusProcessoTest	marcarPreferencialProcesso
10	TC11_ManterCadastroSustentacao-OralTest	alterarPresencaInscricaoPara-SustentacaoOral

a priorização utilizando o algoritmo guloso total foi de 17 milissegundos e o algoritmo guloso adicional foi de 181 milissegundos.

5.2.3 Avaliação dos resultados obtidos

Com as ordens de execução calculadas para cada uma das abordagens sendo avaliadas e as informações a respeito da capacidade de detecção de falhas para cada caso de teste disponível, utilizou-se a fórmula para o cálculo da APFD para avaliar os resultados obtidos. Novamente, para as ordens randômicas optou-se por calcular o valor da APFD para cada execução e depois é realizado o cálculo da média aritmética dessas execuções.

Na Figura 5.6 pode-se visualizar os resultados obtidos com a abordagem proposta e com as execuções não ordenadas dos casos de teste.

Aplicando a fórmula da APFD para cada uma das ordens de execução obtidas, a execução não ordenada dos casos de teste por meio da ordem original obteve o menor valor para a APFD com 50,60%. Um pouco melhor, a média das execuções de ordens randômicas obteve uma APFD de 57,35%.

Entre as ordens utilizando a abordagem proposta nesta pesquisa, a ordem de execução gerada com o algoritmo guloso total obteve um resultado pior que as execuções de ordens randômicas com uma APFD de 52,58%. Entretanto, a ordem de execução gerada com o algoritmo guloso adicional obteve o melhor resultado deste estudo de caso com uma APFD de 72,34%.

5.3 Discussão

Após o término dos dois estudos de caso, algumas informações relevantes podem ser obtidas com a comparação entre os estudos de caso e também com estudos anteriores nesta

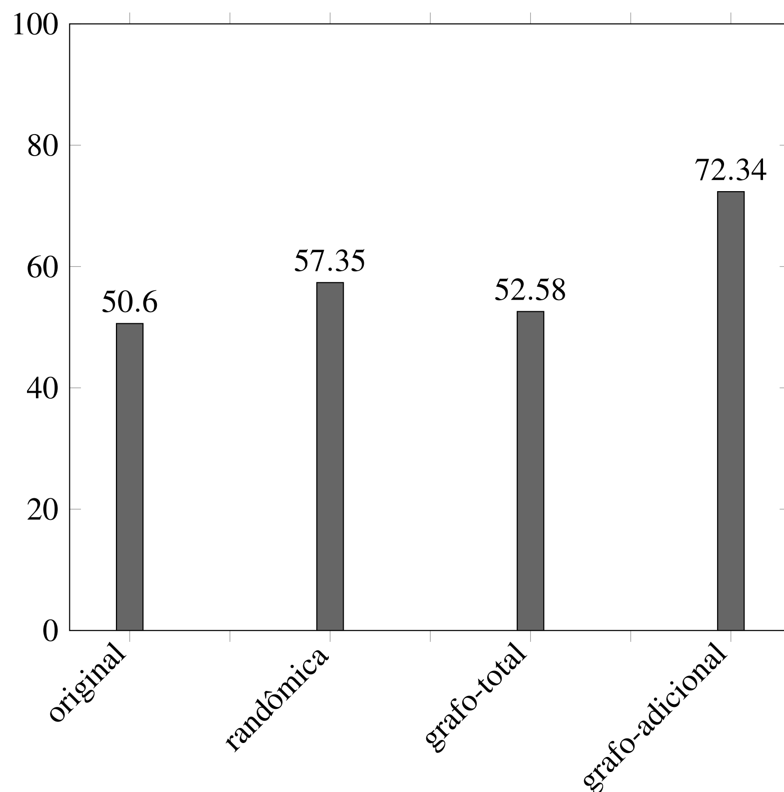


Figura 5.6: APFD para as ordens de execução avaliadas no segundo estudo de caso

área de pesquisa. Ao comparar os resultados obtidos com as execuções não ordenadas dos casos de teste (ordem original e ordens randômicas) e também com a abordagem tradicional de priorização (cobertura de código por instrução) neste trabalho com estudos anteriores que utilizaram a métrica APFD, verifica-se que os resultados obtidos são muito similares. As execuções não ordenadas costumam alcançar valores inferiores a 60% e as execuções priorizadas com a abordagem tradicional alcançam valores superiores a 70%, chegando a mais de 90% com o algoritmo guloso adicional (DO; ROTHERMEL, 2006; MEI et al., 2012).

Pode-se afirmar também que a abordagem proposta utilizando o algoritmo guloso adicional alcançou resultados promissores em relação à execução não ordenada dos casos de teste de sistema automatizados. Ao comparar-se os valores da APFD obtidos com a abordagem proposta e com a ordem original de execução, os ganhos com a nova abordagem ficam entre 20% e 60%. E quando são comparados os valores obtidos com a abordagem proposta e a ordem randômica de execução, os ganhos ficam entre 15% e 30% (ver Figuras 5.5 e 5.6).

Quando avalia-se o tempo necessário para execução da abordagem proposta em relação à execução não ordenada, também é possível dizer que o uso da abordagem proposta é benéfico. Considerando-se que o tempo gasto na priorização dos casos de teste (criação dos grafos de chamadas e ordenamento) no primeiro estudo foi de aproximadamente 30 segundos, a execução priorizada dos casos de teste permitiu que todas as falhas fossem detectadas após a execução de 17 dos 97 casos de teste.

Isto quer dizer que a execução priorizada dos casos de teste levaria apenas 1 minuto e 45 segundos aproximadamente para detectar todas as falhas naquela versão. Por outro lado,

a ordem original de execução encontraria as mesmas falhas somente após 89 casos de teste, ou seja, após 9 minutos e 10 segundos aproximadamente.

Da mesma forma, os 35 segundos necessários para priorizar os casos de teste no segundo estudo de caso são justificáveis, haja visto que todas as falhas são encontradas pelo conjunto de casos de teste priorizado após a execução de 76 dos 97 casos de teste, o que levaria 54 minutos e 51 segundos aproximadamente. Já na ordem original, são necessários 89 casos de teste para encontrar as mesmas falhas e isto elevaria o tempo de execução para 1 hora, 4 minutos e 13 segundos aproximadamente.

Ao comparar o tempo utilizado para priorização dos casos de teste e o tamanho médio dos grafos de chamada para os casos de teste em ambos os estudos, verificou-se que esta solução mesmo utilizando algoritmos lentos consegue suportar bem a escalabilidade. Com uma diferença na ordem de 5 vezes no tamanho dos casos de teste entre os estudos de caso (32 nós em média no primeiro estudo e 169 nós no segundo estudo) não houve grande impacto em relação ao tempo para priorização (acréscimo de apenas 5 segundos).

E considerando que casos de teste de sistema automatizados podem levar várias horas ou até dias para serem executados, esses poucos segundos ou mesmo minutos utilizados para a priorização dos casos de teste não teriam grande impacto sobre o tempo total de execução do conjunto de casos de testes. Porém, como observado em ambos os estudos, podem alcançar ganhos interessantes na taxa de detecção de falhas.

Na comparação entre os dois algoritmos estudados utilizando a abordagem proposta, os dois estudos de caso mostraram que o algoritmo guloso total não consegue obter resultados tão promissores quanto o algoritmo guloso adicional. Após avaliar as ordens de execução geradas por estes dois algoritmos, pode-se afirmar que este comportamento diferenciado justifica-se pela ineficácia do algoritmo guloso total em mesclar a execução de casos de teste de funcionalidades diferentes durante a priorização dos casos de teste.

Ao selecionar os casos de teste apenas pelo seu tamanho e ignorar métodos já executados anteriormente, o algoritmo guloso total acaba por priorizar casos de teste muito similares e isto acaba causando uma diminuição na efetividade do conjunto de casos de teste priorizado. Isto pode ser facilmente visualizado na Tabela 5.6 do primeiro estudo de caso, no qual apenas 2 das 5 opções de menu são testadas nos primeiros 10 casos de teste executados.

Já no caso do algoritmo guloso adicional para o mesmo conjunto de teste, pelo menos um caso de teste para cada uma das 5 opções de menu é executado já nos primeiros 6 casos de teste priorizados (ver Tabela 5.7). Da mesma forma, no segundo estudo de caso, enquanto apenas 4 das 15 funcionalidades (classes de teste) tem casos priorizados para execução nos primeiro 10 casos de teste (ver Tabela 5.12), o guloso adicional selecionou 8 classes diferentes já nos primeiros 8 casos de teste priorizados (ver Tabela 5.13).

É importante destacar também que estes resultados estão fortemente relacionados ao tamanho dos grafos gerados para os casos de teste e a forma como o código de teste foi escrito. Quanto mais complexos os grafos de chamadas gerados e maior a reutilização de código entre os casos de teste, maiores as chances do algoritmo guloso adicional conseguir priorizar casos de teste diferentes entre si para serem executados. Da mesma forma, quanto menores os grafos de chamadas e menor a reutilização de código, maiores as chances de empates durante a priorização, obrigando o algoritmo a selecionar randomicamente o próximo caso de teste a ser priorizado.

Ao comparar a abordagem proposta utilizando o algoritmo guloso adicional a abordagem tradicional utilizando o algoritmo guloso total ou o guloso adicional, verificam-se ganhos

similares entre as duas abordagens. Quando comparadas no primeiro estudo de caso, a abordagem proposta consegue ser apenas 4,95% melhor que o guloso total utilizando cobertura de código por instruções e 15,67% melhor que o guloso adicional (ver Figura 5.5).

Entretanto, é importante ressaltar a diferença entre as abordagens na sua aplicação. Enquanto a abordagem proposta não precisa de nenhuma informação adicional além do código-fonte dos próprios casos de teste, a abordagem tradicional necessita que os dados de cobertura da aplicação sendo testada sejam capturados para cada caso de teste antes da priorização.

Entre as limitações deste trabalho, tem-se principalmente as características relacionadas as aplicações sendo testadas e os casos de teste escritos para estas aplicações. Como ambos os estudos baseiam-se em aplicações escritas em Java e seus casos de teste utilizam bibliotecas e ferramentas para esta linguagem, não é possível neste momento, estender os resultados obtidos para qualquer cenário de testes envolvendo casos de teste de sistema automatizados.

Uma ameaça à validade relevante está relacionada as ferramentas utilizadas para criação dos grafos de chamadas e do cálculo de cobertura de código por instruções. Apesar de uma verificação visual dos resultados obtidos pelas ferramentas ter sido executada durante a pesquisa, o uso de ferramentas diferentes ou mesmo erros nas ferramentas utilizadas poderiam causar diferenças nos resultados obtidos neste trabalho.

Outra ameaça à validade presente no primeiro estudo de caso está relacionada a seleção de casos de teste. Neste estudo de caso, optou-se por não utilizar um critério de teste para a seleção dos casos de teste, isto porque poderia ser selecionada uma grande quantidade de casos de teste, impossibilitando a automação de todos eles durante o desenvolvimento deste trabalho. Assim sendo, o conjunto de casos de teste usado no estudo de caso pode não ter sido o que tinha maior probabilidade de revelar defeitos.

Em relação ao uso dos grafos de chamadas, uma ameaça à validade, que também está presente em trabalhos anteriores, ocorre ao ignorar-se os métodos de configuração (anotações @*Before* e @*BeforeClass* de JUnit) e os métodos de desconfiguração (anotações @*After* e @*AfterClass* de JUnit) na criação do grafo de chamadas para cada um dos casos de teste. Desta forma, o grafo de chamadas representa apenas o método principal de teste (anotação @*Test* de JUnit) e isto pode em alguns casos não representar completamente o cenário de teste sendo verificado pelo caso de teste (ZHANG et al., 2009; MEI et al., 2012).

5.4 Considerações finais

Neste capítulo foram apresentados os dois estudos de caso executados neste trabalho, assim como, os resultados obtidos nestes estudos. Fez-se também uma discussão sobre estes resultados, além de serem descritas as limitações e as ameaças a sua validade. Entre os resultados mais relevantes, está a possibilidade de ganho real na taxa de detecção de falhas a partir do uso da abordagem proposta usando o algoritmo guloso adicional e também os resultados menos promissores desta mesma abordagem usando o algoritmo guloso total. No próximo capítulo são apresentadas as conclusões deste trabalho e as lacunas para trabalhos futuros.

Capítulo 6

Conclusões

Em busca de dar maior agilidade a atividade de testes, o uso da priorização de casos de teste visa diminuir o tempo necessário para detectar as falhas na aplicação sendo desenvolvida por meio do ordenamento dos casos de teste antes da sua execução. Por sua vez, os testes de sistema automatizados permitem reduzir o tempo necessário para execução dos testes mediante a criação de programas que executam os casos de teste sem a necessidade de intervenção humana.

Este trabalho propôs e avaliou uma abordagem para a priorização de casos de teste de sistema automatizados por meio de grafos de chamadas dos próprios casos de teste. E a partir da execução de dois estudos de caso concluiu-se que a utilização desta abordagem permitiu a detecção mais rápida das falhas presentes na aplicação sendo testada.

Para avaliar se a abordagem proposta é capaz de aumentar a taxa de detecção de falhas para casos de teste de sistema automatizados, a métrica APFD foi utilizada para calcular os ganhos obtidos pela abordagem em relação à execução não ordenada dos casos de teste. E em ambos os estudos de caso, verificaram-se ganhos com o uso da abordagem proposta na taxa de detecção de falhas.

Em relação a outras abordagens já estudadas na literatura, avaliou-se a diferença entre os resultados obtidos por esta nova abordagem e a abordagem tradicional de priorização por cobertura de código. Concluiu-se por meio desta avaliação que ambas abordagens conseguiram ganhos similares na taxa de detecção de falhas para o estudo de caso executado.

Para avaliar o impacto da escolha do algoritmo de priorização nos resultados obtidos por esta nova abordagem, ambos estudos de caso compararam a utilização de dois algoritmos comuns de priorização: o guloso total e o guloso adicional. Esta avaliação permitiu concluir que para a abordagem proposta, o uso do guloso adicional obteve resultados promissores que justificam sua utilização, enquanto o uso do guloso total não apresentou ganhos significativos em relação a execução não ordenada dos casos de teste.

Entre as limitações presentes neste trabalho, é importante dizer que os resultados obtidos nesta pesquisa não podem ser estendidos a qualquer cenário de teste que utilize casos de teste de sistema automatizados, pois a abordagem proposta está diretamente relacionada a forma como os casos de teste são implementados. Um cenário em que cada caso de teste é escrito independentemente dos outros casos de teste (sem reaproveitamento de código, por exemplo) possivelmente não se beneficiaria do uso desta abordagem.

Portanto, uma das lacunas desta pesquisa que poderia ser estudada em trabalho futuros é a aplicação desta abordagem em outros cenários de teste, utilizando linguagens e paradigmas

de programação diferentes. Outra limitação deste trabalho que pode ser estudada em trabalhos futuros é a necessidade de um estudo mais aprofundado em relação ao impacto das bibliotecas de teste nos resultados obtidos com esta e outras abordagens. Como por exemplo, a utilização das anotações de configuração e desconfiguração de JUnit na criação do grafo de chamadas.

Outro trabalho futuro para esta área de pesquisa é estudar o estado atual de utilização da priorização de casos de teste pela comunidade e o impacto do seu uso. Mesmo após uma extensa revisão bibliográfica é difícil dizer se e quando esta técnica tem sido utilizada e com que resultados.

Por fim, as contribuições deste trabalho para a área de pesquisa em priorização de casos de teste podem ser listadas como: o mapeamento dos estudos até o ano de 2014 em cinco diferentes categorias, a implementação dos algoritmos guloso total e guloso adicional para priorização de casos de teste por cobertura de código ou por grafos de chamadas, a disponibilização dos casos de teste de sistema criados para a aplicação JMeter, e a avaliação do uso dos grafos de chamadas na priorização de casos de teste de sistema em relação à execução não ordenada destes e à priorização utilizando cobertura de código da aplicação sendo testada.

Referências Bibliográficas

AGGRAWAL, K. K.; SINGH, Y.; KAUR, A. Code coverage based technique for prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 5, p. 1–4, set. 2004. Disponível em: <http://dx.doi.org/10.1145/1022494.1022511>.

ALI, K.; LHOTÁK, O. Averroes: Whole-program analysis without the whole program. In: CASTAGNA, G. (Ed.). *ECOOP 2013 - Object-Oriented Programming*. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 7920). p. 378–400. Disponível em: <http://dx.doi.org/10.1007/978-3-642-39038-8_16>.

ANDREWS, J.; BRIAND, L.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? [software testing]. In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. [s.n.], 2005. p. 402–411. Disponível em: <http://dx.doi.org/10.1109/ICSE.2005.1553583>.

BAI, X.; LAM, C.; LI, H. An approach to generate the thin-threads from the uml diagrams. In: *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. [s.n.], 2004. p. 546–552 vol.1. Disponível em: <http://dx.doi.org/10.1109/CMPSAC.2004.1342893>.

BELLI, F.; EMINOV, M.; GOKCE, N. Prioritizing coverage-oriented testing process - an adaptive-learning-based approach and case study. In: *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. [s.n.], 2007. v. 2, p. 197–203. Disponível em: <http://dx.doi.org/10.1109/COMPSAC.2007.169>.

BRYCE, R. C.; COLBOURN, C. J. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, v. 48, n. 10, p. 960 – 970, 2006. Advances in Model-based Testing. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2006.03.004>.

BRYCE, R. C.; MEMON, A. M. Test suite prioritization by interaction coverage. In: *Workshop on Domain Specific Approaches to Software Test Automation: In Conjunction with the 6th ESEC/FSE Joint Meeting*. New York, NY, USA: ACM, 2007. (DOSTA '07), p. 1–7. Disponível em: <http://dx.doi.org/10.1145/1294921.1294922>.

CAPGEMINI; HEWLETT-PACKARD; SOGETI. *World Quality Report 2015-16*. [S.l.], 2015. Disponível em: <https://www.capgemini.com/resources/world-quality-report-2015-16>, último acesso em 23/02/2016.

CARLSON, R.; DO, H.; DENTON, A. A clustering approach to improving test case prioritization: An industrial case study. In: *Software Maintenance (ICSM), 2011*

27th IEEE International Conference on. [s.n.], 2011. p. 382–391. Disponível em: <http://dx.doi.org/10.1109/ICSM.2011.6080805>.

CATAL, C.; MISHRA, D. Test case prioritization: a systematic mapping study. *Software Quality Journal*, Springer US, v. 21, n. 3, p. 445–478, 2013. Disponível em: <http://dx.doi.org/10.1007/s11219-012-9181-z>.

CHEN, X. et al. Building prioritized pairwise interaction test suites with ant colony optimization. In: *Quality Software, 2009. QSIC '09. 9th International Conference on*. [s.n.], 2009. p. 347–352. Disponível em: <http://dx.doi.org/10.1109/QSIC.2009.52>.

DELAMARO, M.; MALDONADO, J.; JINO, M. *Introdução ao teste de software*. [S.l.]: CAMPUS - RJ, 2007.

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, Springer, v. 10, n. 4, p. 405–435, 2005. Disponível em: <http://dx.doi.org/10.1007/s10664-005-3861-2>.

DO, H. et al. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, v. 36, n. 5, p. 593–617, 2010. Disponível em: <http://dx.doi.org/10.1109/TSE.2010.58>.

DO, H.; ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Software Engineering, IEEE Transactions on*, v. 32, n. 9, p. 733–752, Sept 2006. Disponível em: <http://dx.doi.org/10.1109/TSE.2006.92>.

DO, H.; ROTHERMEL, G. Using sensitivity analysis to create simplified economic models for regression testing. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2008. (ISSTA '08), p. 51–62. Disponível em: <http://dx.doi.org/10.1145/1390630.1390639>.

DUSTIN, E.; RASHKA, J.; PAUL, J. *Automated software testing: introduction, management, and performance*. [S.l.]: Addison-Wesley Professional, 1999.

ELBAUM, S.; MALISHEVSKY, A.; ROTHERMEL, G. Incorporating varying test costs and fault severities into test case prioritization. In: *Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001. (ICSE '01), p. 329–338. Disponível em: <http://dx.doi.org/10.1109/ICSE.2001.919106>.

ELBAUM, S.; MALISHEVSKY, A.; ROTHERMEL, G. Test case prioritization: a family of empirical studies. *Software Engineering, IEEE Transactions on*, v. 28, n. 2, p. 159–182, Feb 2002. Disponível em: <http://dx.doi.org/10.1109/32.988497>.

ELBAUM, S.; MALISHEVSKY, A. G.; ROTHERMEL, G. Prioritizing test cases for regression testing. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2000. (ISSTA '00), p. 102–112. Disponível em: <http://dx.doi.org/10.1145/347324.348910>.

ELBAUM, S.; ROTHERMEL, G.; PENIX, J. Techniques for improving regression testing in continuous integration development environments. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2014. (FSE 2014), p. 235–245. Disponível em: <http://dx.doi.org/10.1145/2635868.2635910>.

GROVE, D. et al. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, ACM, v. 32, n. 10, p. 108–124, 1997. Disponível em: <http://dx.doi.org/10.1145/263698.264352>.

HALL, M. W.; KENNEDY, K. Efficient call graph analysis. *ACM Lett. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 1, n. 3, p. 227–242, set. 1992. Disponível em: <http://dx.doi.org/10.1145/151640.151643>.

HLA, K.; CHOI, Y.; PARK, J. S. Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting. In: *Computer and Information Technology Workshops, 2008. CIT Workshops 2008. IEEE 8th International Conference on*. [s.n.], 2008. p. 527–532. Disponível em: <http://dx.doi.org/10.1109/CIT.2008.Workshops.104>.

JIANG, B.; CHAN, W. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In: *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. [s.n.], 2013. p. 190–199. Disponível em: <http://dx.doi.org/10.1109/COMPSAC.2013.33>.

JIANG, B. et al. Adaptive random test case prioritization. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ASE '09), p. 233–244. Disponível em: <http://dx.doi.org/10.1109/ASE.2009.77>.

JONES, J.; HARROLD, M. Test-suite reduction and prioritization for modified condition/decision coverage. *Software Engineering, IEEE Transactions on*, v. 29, n. 3, p. 195–209, March 2003. Disponível em: <http://dx.doi.org/10.1109/TSE.2003.1183927>.

KAPFHAMMER, G. M.; SOFFA, M. L. Using coverage effectiveness to evaluate test suite prioritizations. In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. New York, NY, USA: ACM, 2007. (WEASELTech '07), p. 19–20. Disponível em: <http://dx.doi.org/10.1145/1353673.1353677>.

KAUFFMAN, J.; KAPFHAMMER, G. A framework to support research in and encourage industrial adoption of regression testing techniques. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. [s.n.], 2012. p. 907–908. Disponível em: <http://dx.doi.org/10.1109/ICST.2012.194>.

KIM, J.-M.; PORTER, A. A history-based test prioritization technique for regression testing in resource constrained environments. In: IEEE. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*. 2002. p. 119–129. Disponível em: <http://dx.doi.org/10.1109/ICSE.2002.1007961>.

KINNEER, A.; DWYER, M. B.; ROTHERMEL, G. Sofya: A flexible framework for development of dynamic program analyses for java software. *CSE Technical reports*, p. 20, 2006. Disponível em: <http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1022&context=csetechreports>.

KOREL, B.; TAHAT, L.; HARMAN, M. Test prioritization using system models. In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [s.n.], 2005. p. 559–568. Disponível em: <http://dx.doi.org/10.1109/ICSM.2005.87>.

LAM, P. et al. The soot framework for java program analysis: a retrospective. In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. [s.n.], 2011. Disponível em: <https://sable.github.io/soot/resources/lblh11soot.pdf>.

LEON, D.; PODGURSKI, A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. [s.n.], 2003. p. 442–453. Disponível em: <http://dx.doi.org/10.1109/ISSRE.2003.1251065>.

LI, S. et al. A simulation study on some search algorithms for regression test case prioritization. In: *Quality Software (QSIC), 2010 10th International Conference on*. [s.n.], 2010. p. 72–81. ISSN 1550-6002. Disponível em: <http://dx.doi.org/10.1109/QSIC.2010.15>.

LI, Z.; HARMAN, M.; HIERONS, R. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, v. 33, n. 4, p. 225–237, 2007. Disponível em: <http://dx.doi.org/10.1109/TSE.2007.38>.

LIMA, L. et al. Test case prioritization based on data reuse an experimental study. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Washington, DC, USA: IEEE Computer Society, 2009. (ESEM '09), p. 279–290. Disponível em: <http://dx.doi.org/10.1109/ESEM.2009.5315980>.

LIN, C.-T. et al. History-based test case prioritization with software version awareness. In: *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*. [s.n.], 2013. p. 171–172. Disponível em: <http://dx.doi.org/10.1109/ICECCS.2013.33>.

LV, J.; YIN, B.; CAI, K.-Y. On the gain of measuring test case prioritization. In: *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. [s.n.], 2013. p. 627–632. Disponível em: <http://dx.doi.org/10.1109/COMPSAC.2013.101>.

MA, Z.; ZHAO, J. Test case prioritization based on analysis of program structure. In: *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*. [s.n.], 2008. p. 471–478. Disponível em: <http://dx.doi.org/10.1109/APSEC.2008.63>.

MALISHEVSKY, A.; ROTHERMEL, G.; ELBAUM, S. Modeling the cost-benefits tradeoffs for regression testing techniques. In: *Software Maintenance, 2002. Proceedings. International Conference on*. [s.n.], 2002. p. 204–213. Disponível em: <http://dx.doi.org/10.1109/ICSM.2002.1167767>.

MARIANI, L.; PAPAGIANNAKIS, S.; PEZZÈ, M. Compatibility and regression testing of cots-component-based software. In: *Software Engineering, 2007. ICSE*

2007. 29th International Conference on. [s.n.], 2007. p. 85–95. Disponível em: <http://dx.doi.org/10.1109/ICSE.2007.26>.

MARIJAN, D.; GOTLIEB, A.; SEN, S. Test case prioritization for continuous regression testing: An industrial case study. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [s.n.], 2013. p. 540–543. Disponível em: <http://dx.doi.org/10.1109/ICSM.2013.91>.

MEI, H. et al. A static approach to prioritizing junit test cases. *Software Engineering, IEEE Transactions on*, v. 38, n. 6, p. 1258–1275, Nov 2012. Disponível em: <http://dx.doi.org/10.1109/TSE.2011.106>.

MEI, L. et al. Xml-manipulating test case prioritization for xml-manipulating services. *Journal of Systems and Software*, Elsevier, v. 84, n. 4, p. 603–619, 2011. Disponível em: <http://dx.doi.org/10.1016/j.jss.2010.11.905>.

MIRARAB, S.; TAHVILDARI, L. An empirical study on bayesian network-based approach for test case prioritization. In: *Software Testing, Verification, and Validation, 2008 1st International Conference on*. [s.n.], 2008. p. 278–287. Disponível em: <http://dx.doi.org/10.1109/ICST.2008.57>.

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

PARK, H.; RYU, H.; BAIK, J. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In: *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*. [s.n.], 2008. p. 39–46. Disponível em: <http://dx.doi.org/10.1109/SSIRI.2008.52>.

QU, X.; COHEN, M.; WOOLF, K. Combinatorial interaction regression testing: A study of test case generation and prioritization. In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. [s.n.], 2007. p. 255–264. Disponível em: <http://dx.doi.org/10.1109/ICSM.2007.4362638>.

RAFI, D. M. et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: IEEE PRESS. *Proceedings of the 7th International Workshop on Automation of Software Test*. 2012. p. 36–42. Disponível em: <http://dx.doi.org/10.1109/IWAST.2012.6228988>.

RAMANATHAN, M. K. et al. Phalanx: A graph-theoretic framework for test case prioritization. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2008. (SAC '08), p. 667–673. Disponível em: <http://dx.doi.org/10.1145/1363686.1363848>.

ROTHERMEL, G. et al. Test case prioritization: an empirical study. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. [s.n.], 1999. p. 179–188. Disponível em: <http://dx.doi.org/10.1109/ICSM.1999.792604>.

SAMPATH, S. et al. Prioritizing user-session-based test cases for web applications testing. In: *Software Testing, Verification, and Validation, 2008 1st International Conference on*. [s.n.], 2008. p. 141–150. Disponível em: <http://dx.doi.org/10.1109/ICST.2008.42>.

SHERRIFF, M.; LAKE, M.; WILLIAMS, L. Prioritization of regression tests using singular value decomposition with empirical change records. In: *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*. [s.n.], 2007. p. 81–90. Disponível em: <http://dx.doi.org/10.1109/ISSRE.2007.25>.

SRIKANTH, H.; BANERJEE, S. Improving test efficiency through system test prioritization. *Journal of Systems and Software*, v. 85, n. 5, p. 1176 – 1187, 2012. Disponível em: <http://dx.doi.org/10.1016/j.jss.2012.01.007>.

SRIKANTH, H.; COHEN, M.; QU, X. Reducing field failures in system configurable software: Cost-based prioritization. In: *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*. [s.n.], 2009. p. 61–70. Disponível em: <http://dx.doi.org/10.1109/ISSRE.2009.26>.

SRIKANTH, H.; WILLIAMS, L.; OSBORNE, J. System test case prioritization of new and regression test cases. In: *Empirical Software Engineering, 2005. 2005 International Symposium on*. [s.n.], 2005. p. 10 pp.–. Disponível em: <http://dx.doi.org/10.1109/ISESE.2005.1541815>.

STAATS, M.; LOYOLA, P.; ROTHERMEL, G. Oracle-centric test case prioritization. In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. [s.n.], 2012. p. 311–320. Disponível em: <http://dx.doi.org/10.1109/ISSRE.2012.13>.

TIKIR, M. M.; HOLLINGSWORTH, J. K. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 27, n. 4, p. 86–96, 2002. Disponível em: <http://dx.doi.org/10.1145/566172.566186>.

TONELLA, P.; AVESANI, P.; SUSI, A. Using the case-based ranking methodology for test case prioritization. In: *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*. [s.n.], 2006. p. 123–133. Disponível em: <http://dx.doi.org/10.1109/ICSM.2006.74>.

WANG, S. et al. Multi-objective test prioritization in software product line testing: An industrial case study. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. New York, NY, USA: ACM, 2014. (SPLC '14), p. 32–41. Disponível em: <http://dx.doi.org/10.1145/2648511.2648515>.

WAZLAWICK, R. *Metodologia de Pesquisa para Ciência da Computação, 2ª Edição*. [S.l.]: Elsevier Brasil, 2014. v. 2.

WONG, W. et al. A study of effective regression testing in practice. In: *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. [s.n.], 1997. p. 264–274. Disponível em: <http://dx.doi.org/10.1109/ISSRE.1997.630875>.

XU, D.; DING, J. Prioritizing state-based aspect tests. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. [s.n.], 2010. p. 265–274. Disponível em: <http://dx.doi.org/10.1109/ICST.2010.14>.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, Wiley Online Library, v. 22, n. 2, p. 67–120, 2012. Disponível em: <http://dx.doi.org/10.1002/stvr.430>.

YU, Y.; NG, S.; CHAN, E. Generating, selecting and prioritizing test cases from specifications with tool support. In: *Quality Software, 2003. Proceedings. Third International Conference on*. [s.n.], 2003. p. 83–90. Disponível em: <http://dx.doi.org/10.1109/QSIC.2003.1319089>.

ZHAI, K.; JIANG, B.; CHAN, W. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *Services Computing, IEEE Transactions on*, v. 7, n. 1, p. 54–67, Jan 2014. Disponível em: <http://dx.doi.org/10.1109/TSC.2012.40>.

ZHAI, K. et al. Taking advantage of service selection: A study on the testing of location-based web services through test case prioritization. In: *Web Services (ICWS), 2010 IEEE International Conference on*. [s.n.], 2010. p. 211–218. Disponível em: <http://dx.doi.org/10.1109/ICWS.2010.98>.

ZHANG, L. et al. Prioritizing junit test cases in absence of coverage information. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. [s.n.], 2009. p. 19–28. Disponível em: <http://dx.doi.org/10.1109/ICSM.2009.5306350>.

ZHI-HUA, Z.; YONG-MIN, M.; YING-AI, T. Test case prioritization for regression testing based on function call path. In: *Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on*. [s.n.], 2012. p. 1372–1375. Disponível em: <http://dx.doi.org/10.1109/ICCIS.2012.312>.

Apêndice A

Estudo de mapeamento da área de pesquisa

Este apêndice segue com a Seção A.1 apresentando o método de pesquisa utilizado neste estudo de mapeamento. Na Seção A.2 são apresentados os resultados deste mapeamento.

A.1 Método de pesquisa

O objetivo deste mapeamento sistemático é responder as seguintes perguntas:

1. Quais os critérios já pesquisados como base para PCT?
2. Quais os algoritmos já utilizados na PCT?
3. Quais as métricas já utilizadas na validação dos resultados obtidos com a PCT?
4. Quais os objetos de estudo mais comuns em estudos sobre PCT?

A.1.1 Estratégia de pesquisa

Para buscar por estudos sobre o tema de interesse deste trabalho, o seguinte conjunto de palavras foi utilizado na busca: *software* E *test* E (*prioritized* OU *prioritization* OU *prioritizing*). Na tentativa de garantir que todos os artigos sobre priorização estivessem contidos nesta busca, o intervalo de tempo escolhido para as buscas foi de 1997, ano da publicação do primeiro trabalho citando a técnica de PCT até o final de 2014.

A fim de garantir uma boa variedade de fontes eletrônicas, as bases de dados usadas neste trabalho cobrem os jornais, conferências e workshops relevantes na área de engenharia de software:

- ACM Digital Library (portal.acm.org).
- IEEE eXplore (ieeexplore.ieee.org).
- ScienceDirect (sciencedirect.com).

Ao final da busca inicial, 613 artigos foram encontrados. Após uma leitura dos títulos e resumos dos artigos e remoção de trabalhos duplicados em mais de uma base de pesquisa, 225 destes foram selecionados para serem analisados em detalhes. Ao final da fase de classificação 178 foram catalogados neste estudo, 14 eram trabalhos prévios ou incompletos que foram posteriormente republicados e 33 não tratavam do tema central deste estudo.

A.1.2 Esquema de classificação

Para responder as peguntas levantadas por este estudo, os artigos selecionados foram analisados com base em cinco diferentes categorias: (i) critério usado na priorização, (ii) algoritmo usado na priorização, (iii) métrica de avaliação e (iv) objeto de estudo. É importante salientar que nem todas estas categorias são obrigatórias para que um estudo sobre PCT seja desenvolvido e um mesmo artigo pode trazer estudos com zero ou mais variáveis para uma mesma categoria. Portanto, a soma total de estudo mapeados dentro de cada uma das categorias não necessariamente será igual ao número total de estudos pesquisados.

A.2 Mapeamento

Na Figura A.1 são apresentados os diferentes critérios de priorização já utilizados e a quantidade de estudos a utilizar cada um deles. Os critérios de priorização foram separados em 8 grupos: cobertura de código, dados históricos, potencial de exposição de falhas, modelo do sistema, dados de entrada, requisitos do sistema e vetores de cobertura. Entre estes grupos, a cobertura de código foi o critério mais utilizado, seguido dos dados históricos e dos modelos do sistema.

Na Figura A.2 encontra-se a distribuição dos estudos entre os grupos de algoritmos já estudados nesta área de pesquisa. Os algoritmos de priorização foram separados em 6 grupos: gulosos, busca local, evolucionários, aprendizado de máquina, híbridos e outros. Entre estes grupos, os algoritmos gulosos se destacam por serem os mais utilizados.

Na Figura A.3 é apresentada a distribuição dos estudos mapeados entre as métricas de avaliação propostas. As métricas de avaliação para estudos sobre priorização foram separadas em 12 grupos: APFD, $APFD_c$, baseadas na APFD, HMFD, detecção de falhas, *relative position*, *coverage effectiveness*, cobertura de código, tempo de execução, modelos econômicos, *interaction coverage* e outras. Entre estes grupos, a métrica APFD destaca-se por ser a mais utilizada.

Na Figura A.4 é apresentada a distribuição de todos os objetos de estudo usados nos estudos mapeados. Os objetos de estudo foram separados em 9 grupos: *Space*, *Siemens Suite*, aplicações em C de código aberto (SIR), aplicações em Java de código aberto (SIR), aplicações web de código aberto, exemplos/modelos de aplicações simples, aplicações universitárias, aplicações industriais e outras aplicações. Entre estes grupos, além do grande número de aplicações simples sendo utilizadas nos estudos mapeados, destaca-se também o grande número de utilizações das aplicações obtidas no SIR.

Na Tabela A.1 são listados todos os estudos mapeados assim como a classificação adotada para cada um deles dentro das quatro diferentes categorias analisadas. Na Tabela A.2 são apresentadas as informações sobre os estudos, tais como título, autores e ano de publicação. Para reduzir o espaço necessário na classificação de cada um dos trabalhos, o seguinte conjunto de abreviações foi utilizado em cada uma das categorias:

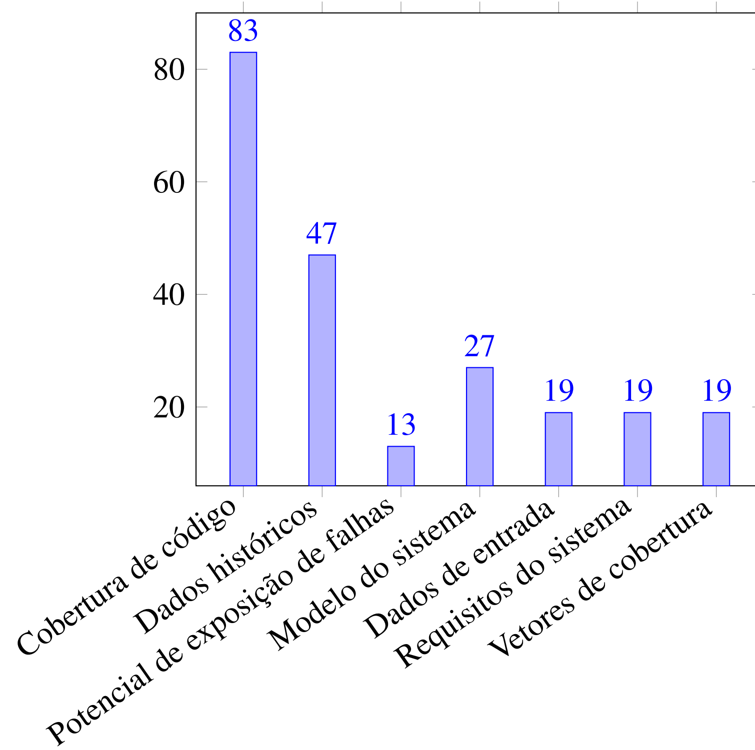


Figura A.1: Distribuição das pesquisas por critério para a priorização

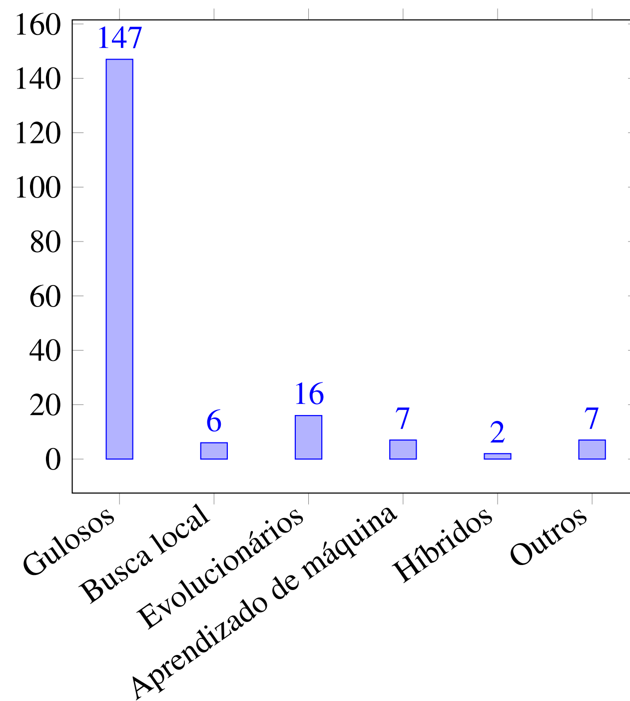


Figura A.2: Distribuição dos algoritmos utilizados para a priorização

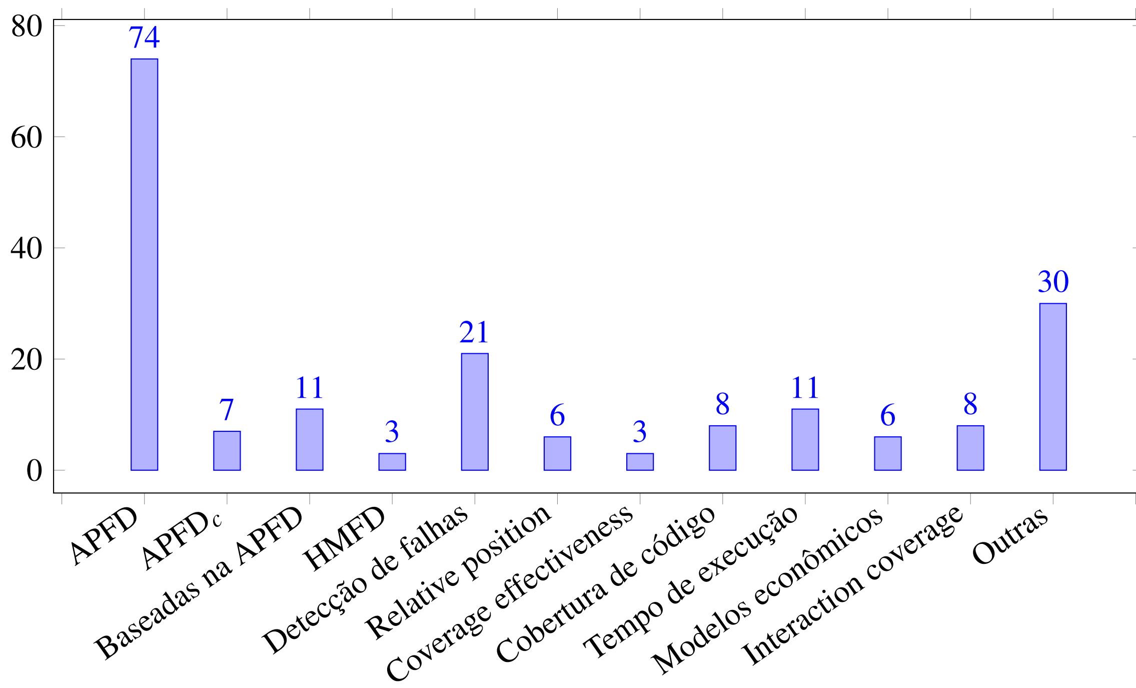


Figura A.3: Distribuição das métricas de avaliação utilizadas na priorização

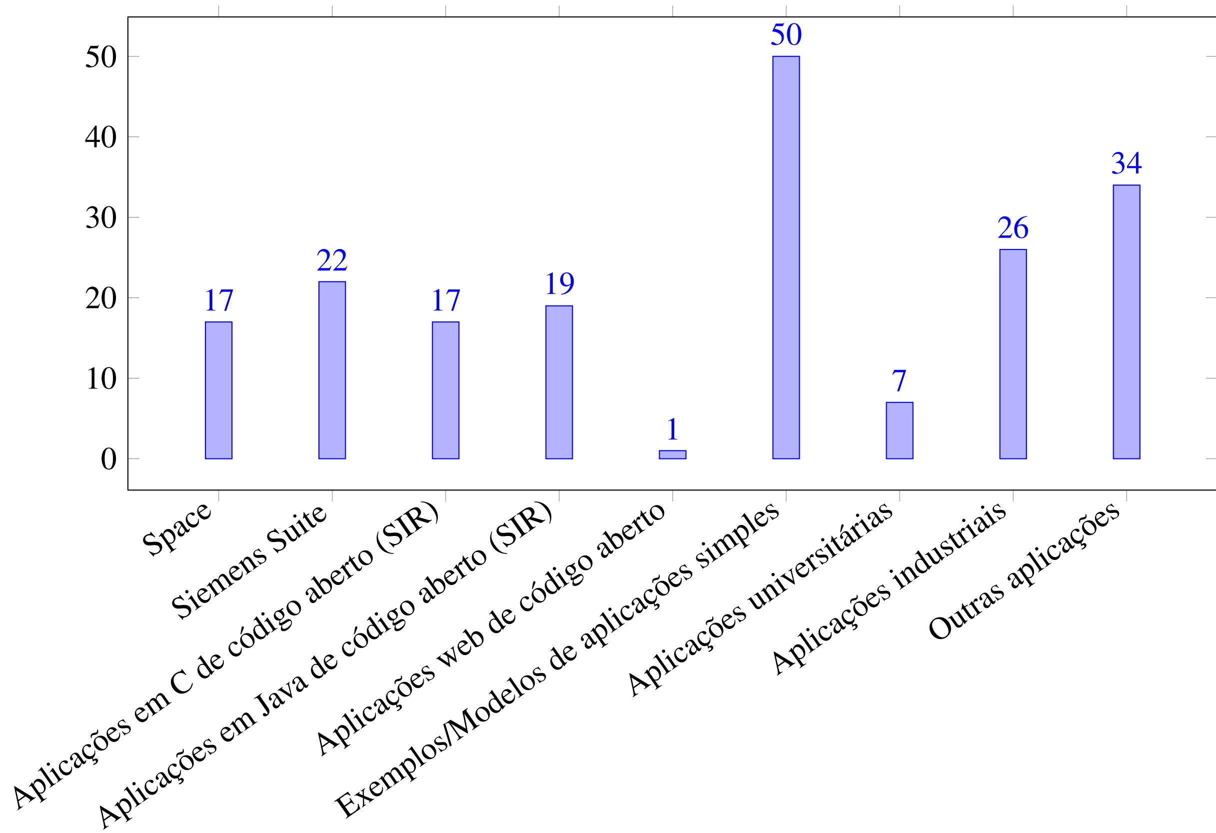


Figura A.4: Distribuição dos objetos de estudo utilizadas na priorização

- Critério

  CC Cobertura de código
  DH Dados históricos
  PEF Potencial de exposição de falhas
  DE Dados de entrada
  RS Requisitos do sistema
  MS Modelos de sistema
  VC Vetores de cobertura

- Algoritmo

  GU Guloso
  BL Busca local
  EV Evolucionário
  AM Aprendizado de máquina
  HI Híbrido
  OU Outros

- Métrica

  AP APFD
  AC $APFD_c$
  BA Baseados na APFD
  HM HMFD
  DF Detecção de falhas
  RP *Relative position*
  CE *Coverage effectiveness*
  CO Cobertura de código
  TE Tempo de execução
  ME Modelos econômicos
  IC *Interaction coverage*
  OM Outras

- Objeto de estudo

  PS Programa Space
  SS Siemens Suite
  JS Aplicações em Java do SIR
  CS Aplicações em C do SIR

AW Aplicações Web de código aberto
AI Aplicações industriais
AS Aplicações ou modelos simples
AU Aplicações universitárias
OA Outras aplicações

Tabela A.1: Classificação detalhada dos estudos mapeados

Estudo	Critérios	Algoritmos	Métricas	Objetos de estudo
E1	CC	GU	CO	AS
E2	CC,DH	EV	AP	AS
E3	CC,DH	GU	CO,TE,OM	OA
E4	CC	GU	AP	JS
E5	PEF	GU	DF,TE,OM	AS
E6	CC,RS	GU	AP	AU
E7	DE	GU	AP	OA
E8	PEF,MS,DE	GU	HM,OM	OA
E9	MS	GU	-	AS
E10	PEF	GU	OM	AS
E11	MS	AM	-	AI
E12	CC	GU	DF	OA
E13	DE,RS	AM	-	AI
E14	DH	EV	AP	AI
E15	VC	GU	OM	AS
E16	VC	GU	AP	AU
E17	VC	HI	IC	AS
E18	DE,VC	GU	AP,DF	OA
E19	CC,DH	GU	AP,DF	AI
E20	VC	HI	IC	AS
E21	VC	EV	IC	AS
E22	MS	GU	-	AS
E23	CC	GU	AP	AI
E24	CC	GU	AP	JS
E25	CC	GU	ME	JS
E26	CC	GU	AP	JS
E27	CC	GU	ME	JS
E28	CC	GU,AM	ME	JS
E29	CC,PEF	GU	AP	PS,SS
E30	CC,PEF	GU	AC	PS
E31	CC	GU	AP	PS,SS
E32	CC,PEF	GU	AP	PS,SS,CS
E33	DH	GU	TE	OA
E34	DH	GU	AP	AI

Tabela A.1: continuação

Estudo	Critérios	Algoritmos	Métricas	Objetos de estudo
E35	DH	GU	AP	PS,SS
E36	MS	GU	OM	AS
E37	VC	GU	-	AS
E38	MS	GU	AP	OA
E39	CC,MS	GU	AP	OA
E40	CC	GU	AP	OA
E41	CC,MS	GU	AP	OA
E42	CC	GU	AP	SS
E43	CC	GU	AC	PS,SS,CS
E44	MS	GU,BL	AP,DF	OA
E45	MS	GU	AP,RP	AS
E46	CC	GU	AP	PS,SS,JS
E47	-	-	-	-
E48	RS	GU	AP,OM	AU
E49	CC	EV	CO	AS
E50	CC,DH	GU	-	-
E51	PEF	GU	AP	AS
E52	CC	GU	AP	OA
E53	CC,DH	GU	AP,DF	AI
E54	DH	EV	AC	CS
E55	VC	GU	BA	AS
E56	VC	GU	IC	CS
E57	CC,DH,RS	EV	AP,BA,OM	OA
E58	CC	GU	AP	SS,CS
E59	CC	GU,OU	AP,TE	SS,CS
E60	DE	OU	AP,TE	CS
E61	CC	GU	TE	PS,SS
E62	CC	EV	CO	OA
E63	-	-	CE	-
E64	CC,DH	GU,BL,EV,OU	AP	AS
E65	MS	GU	-	AS
E66	-	-	-	-
E67	RS	GU	OM	AI
E68	DH	GU	OM	AS
E69	CC,DH	GU	AP	PS,SS
E70	CC	GU	-	AS
E71	CC,DH	GU	-	PS,SS
E72	CC,DH	GU	AP	PS,SS
E73	MS	GU	RP	AS
E74	MS	GU	RP	AS
E75	MS	GU	RP	AS
E76	CC,MS	GU	RP	AS

Tabela A.1: continuação

Estudo	Critérios	Algoritmos	Métricas	Objetos de estudo
E77	RS	GU	OM	AU,AI
E78	CC,DH	GU	CO	AI
E79	CC	GU,AM	AP	OA
E80	CC	GU,BL,EV	CO	PS,SS,CS
E81	-	GU,BL,EV	OM	OA
E82	DH,RS	GU	OM	AI
E83	DE	OU	TE	OA
E84	CC,DH	GU	AC	SS
E85	PEF,RS	GU	DF	OA
E86	-	-	BA	-
E87	CC,DH	GU	AC,OM	JS
E88	DH	GU	AP	AS
E89	CC	EV	OM	AS
E90	CC	GU	ME	CS
E91	DH,MS	GU	OM	AI
E92	DH	GU	DF	OA
E93	DE	GU	-	AI
E94	DH	GU	AC,TE	AI
E95	CC	GU	DF	JS
E96	MS	GU	AP	AS
E97	DE	GU	AP	AW
E98	CC	GU	AP	JS
E99	CC	AM	AP	JS
E100	CC	GU	AP	JS
E101	MS	GU	-	AS
E102	CC,DH,RS	EV	AP,CO,TE	SS
E103	CC	GU	DF,ME	JS
E104	PEF	GU	DF	AU
E105	DH	GU	DF	AU
E106	CC,MS	GU	AP	AI
E107	CC	GU	DF	AS
E108	DH	GU	AC	JS
E109	VC	GU	DF,IC	CS
E110	MS	GU	TE	AS
E111	DH	GU	AP	AS
E112	CC,VC	GU	BA	CS
E113	DH	GU	DF	AI
E114	CC,DH,DE,VC	GU	BA	CS
E115	-	GU	BA	AI
E116	-	-	-	-
E117	CC,DE,VC	GU	BA	CS
E118	CC,DH,RS	GU	AP	AS

Tabela A.1: continuação

Estudo	Critérios	Algoritmos	Métricas	Objetos de estudo
E119	CC	GU,OU	AP,OM	SS,CS
E120	DE	GU	DF	AI
E121	CC,PEF	GU	AP	SS
E122	CC,PEF	GU	AP	PS,SS
E123	CC	GU	AP	CS
E124	CC	GU	AP	AS
E125	CC	EV	-	AS
E126	VC	GU	IC	AS
E127	VC	GU	OM	AS
E128	DH,RS	GU	AP	OA
E129	RS	GU	-	AS
E130	DE,VC	GU	AP,DF	OA
E131	-	-	-	-
E132	DE	GU	AP	OA
E133	DH	GU	AP	OA
E134	MS	GU	AP	OA
E135	MS	GU	AP	AS
E136	MS	GU	-	AS
E137	DH	GU	OM	AI
E138	CC,DH	AM	AP	SS
E139	DH	EV	AP	AS
E140	CC,DH	GU	CE	OA
E141	RS	GU	BA	AI
E142	DH,VC	GU	BA,OM	AI
E143	DH	GU	BA	AI
E144	RS	GU	BA	AI
E145	CC	GU	-	AI
E146	RS	GU	-	AS
E147	MS	GU	AP	AU
E148	CC	GU	AP	OA
E149	CC,DH	GU	AP	JS
E150	CC,PEF	GU	AP	AS
E151	CC	AM	AP	PS
E152	DH	GU	AP,TE	AS
E153	CC	EV	AP	AS
E154	DH,VC	BL,EV	DF,IC,OM	AI
E155	CC,DH,PEF,RS	GU	CO	AS
E156	MS	GU	OM	AS
E157	DH,RS	GU,BL	CE	OA
E158	CC	GU	-	PS
E159	CC	GU	AP,DF,ME	JS,OA
E160	VC	GU,OU	IC,OM	CS,OA

Tabela A.1: continuação

Estudo	Critérios	Algoritmos	Métricas	Objetos de estudo
E161	DH,DE	GU	DF,TE	AI
E162	MS	GU	DF,RP	AS
E163	CC	GU	OM	CS
E164	CC,DH	GU	OM	PS,SS
E165	DE	GU	-	AS
E166	DE	GU	AP,OM	OA
E167	CC	GU	AP	OA
E168	CC,VC	GU	OM	AS
E169	DE	GU	HM,OM	OA
E170	DE	GU	AP	OA
E171	DE	GU	HM	OA
E172	DH,RS	GU	OM	-
E173	CC	GU	AP	JS
E174	CC	GU	AP	JS
E175	-	-	OM	AS
E176	CC	GU	AP	JS
E177	CC	GU	-	-
E178	CC	OU	AP,OM	PS,SS

Tabela A.2: Informações sobre os estudos mapeados

Estudo	Autores	Título	Ano de publicação
E1	K. K. Aggrawal et al.	Code coverage based technique for prioritizing test cases for regression testing	2004
E2	A. Ahmed et al.	Software testing suite prioritization using multi-criteria fitness function	2012
E3	S. Alspaugh et al.	Efficient time-aware prioritization with knapsack solvers	2007
E4	E. L. G. Alves et al.	A refactoring-based approach for test case selection and prioritization	2013
E5	A. Ansari et al.	Optimization of test suite-test case in regression test	2013
E6	M. J. Arafeen, H. Do	Test case prioritization using requirements-based clustering	2013
E7	A. Askarunisa et al.	Sequence-based techniques for black-box test case prioritization for composite service testing	2010
E8	A. Askarunisa et al.	Black box test case prioritization techniques for semantic based composite web services using owl-s	2011
E9	B. Athira, P. Samuel	Web services regression test case prioritization	2010
E10	X. Bai et al.	An approach to generate the thin-threads from the uml diagrams	2004

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E11	F. Belli et al.	Prioritizing coverage-oriented testing process - an adaptive-learning-based approach and case study	2007
E12	A. Beszedes et al.	Code coverage-based regression test selection and prioritization in webkit	2012
E13	H. Bhasin, E. Khanna	Neural network based black box testing	2014
E14	H. Bhasin	Cost-priority cognizant regression testing	2014
E15	R. C. Bryce, C. J. Colbourn	Prioritized interaction testing for pairwise coverage with seeding and constraints	2006
E16	R. C. Bryce, A. M. Memon	Test suite prioritization by interaction coverage	2007
E17	R. C. Bryce, C. J. Colbourn	One-test-at-a-time heuristic search for interaction test suites	2007
E18	R. C. Bryce et al.	Developing a single model and test prioritization strategies for event-driven software	2011
E19	R. Carlson et al.	A clustering approach to improving test case prioritization: An industrial case study	2011
E20	X. Chen et al.	A hybrid approach to build prioritized pairwise interaction test suites	2009
E21	X. Chen et al.	Building prioritized pairwise interaction test suites with ant colony optimization	2009
E22	L. Chen et al.	Test case prioritization for web service regression testing	2010
E23	D. Di Nardo et al.	Coverage-based test case prioritisation: An industrial case study	2013
E24	H. Do et al.	Empirical studies of test case prioritization in a junit testing environment	2004
E25	H. Do, G. Rothermel	An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models	
E26	H. Do, G. Rothermel	On the use of mutation faults in empirical assessments of test case prioritization techniques	2006
E27	H. Do, G. Rothermel	Using sensitivity analysis to create simplified economic models for regression testing	2008
E28	H. Do et al.	The effects of time constraints on test case prioritization: A series of controlled experiments	2010
E29	S. Elbaum et al.	Prioritizing test cases for regression testing	2000
E30	S. Elbaum et al.	Incorporating varying test costs and fault severities into test case prioritization	2001
E31	S. Elbaum et al.	Understanding and measuring the sources of variation in the prioritization of regression test suites	2001

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E32	S. Elbaum et al.	Test case prioritization: a family of empirical studies	2002
E33	S. Elbaum et al.	Techniques for improving regression testing in continuous integration development environments	2014
E34	E. Engstroem et al.	Improving regression testing transparency and efficiency with history-based prioritization - an industrial case study	2011
E35	Y. Fazlalizadeh et al.	Prioritizing test cases for resource constraint environments using historical test case performance data	2009
E36	A. Gantait	Test case generation and prioritization from uml models	2011
E37	J. Gao, J. Zhu	Prioritized test generation strategy for pair-wise testing	2009
E38	D. Garg, A. Datta	Test case prioritization due to database changes in web applications	2012
E39	D. Garg et al.	A two-level prioritization approach for regression testing of web applications	2012
E40	D. Garg, A. Datta	Early detection of faults related to database schematic changes	2013
E41	D. Garg, A. Datta	Parallel execution of prioritized test cases for regression testing of web applications	2013
E42	A. Gonzalez -Sanchez et al.	Prioritizing tests for software fault localization	2010
E43	A. Gonzalez -Sanchez et al.	Prioritizing tests for fault localization through ambiguity group reduction	2011
E44	S. Haidry, T. Miller	Using dependency structures for prioritization of functional test suites	2013
E45	X. Han et al.	A heuristic model-based test prioritization method for regression testing	2012
E46	D. Hao et al.	Adaptive test-case prioritization guided by output inspection	2013
E47	M. Harman	Making the case for morto: Multi objective regression test optimization	2011
E48	C. Hettiarachchi et al.	Effective regression testing using requirements and risks	2014
E49	K. Hla et al.	Applying particle swarm optimization to prioritizing test cases for embedded real time software retesting	2008
E50	I. Holden, D. Dalton	Improving testing efficiency using cumulative test analysis	2006
E51	S.-S. Hou et al.	Applying interface-contract mutation in regression testing of component-based software	2007

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E52	S.-S. Hou et al.	Quota-constrained test-case prioritization for regression testing of service-centric systems	2008
E53	S. Huang et al.	An optimized change-driven regression testing selection strategy for binary java applications	2009
E54	Y.-C. Huang et al.	A history-based cost-cognizant test case prioritization technique in regression testing	2012
E55	R. Huang et al.	Prioritizing variable strength covering array	2013
E56	R. Huang et al.	Adaptive random prioritization for interaction test suites	2014
E57	M. Islam et al.	A multi-objective technique to prioritize test cases based on latent semantic indexing	2012
E58	D. Jeffrey, N. Gupta	Experiments with test case prioritization using relevant slices	2008
E59	B. Jiang et al.	Adaptive random test case prioritization	2009
E60	B. Jiang, W. Chan	Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization	2013
E61	J. Jones, M. Harrold	Test-suite reduction and prioritization for modified condition/decision coverage	2003
E62	W. Jun et al.	Test case prioritization technique based on genetic algorithm	2011
E63	G. Kapfhammer, M. L. Soffa	Using coverage effectiveness to evaluate test suite prioritizations	2007
E64	J. Kauffman, G. Kapfhammer	A framework to support research in and encourage industrial adoption of regression testing techniques	2012
E65	P. Kaur et al.	Prioritization of test scenarios derived from uml activity diagram using path complexity	2012
E66	N. Kaushik et al.	Dynamic prioritization in regression testing	2011
E67	R. Kavitha et al.	Requirement based test case prioritization	2010
E68	M. Kayes	Test case prioritization for regression testing based on fault dependency	2011
E69	A. Khalilian et al.	An improved method for test case prioritization by incorporating historical test case data	2012
E70	S. Khan et al.	The impact of test case reduction and prioritization on software testing effectiveness	2009
E71	J.-M. Kim, A. Porter	A history-based test prioritization technique for regression testing in resource constrained environments	2002

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E72	S. Kim, J. Baik	An effective fault aware test case prioritization by incorporating a fault localization technique	2010
E73	B. Korel et al.	Test prioritization using system models	2005
E74	B. Korel et al.	Model-based test prioritization heuristic methods and their evaluation	2007
E75	B. Korel et al.	Application of system models in regression test suite prioritization	2008
E76	B. Korel, G. Koutsogiannakis	Experimental comparison of code-based and model-based test prioritization	2009
E77	R. Krishnamoorthi, S. S. A. Mary	Factor oriented requirement coverage based system test case prioritization of new and regression test cases	2009
E78	S. Kukolj et al.	Selection and prioritization of test cases by combining white-box and black-box testing methods	2013
E79	D. Leon, A. Podgurski	A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases	2003
E80	Z. Li et al.	Search algorithms for regression test case prioritization	2007
E81	S. Li et al.	A simulation study on some search algorithms for regression test case prioritization	2010
E82	B. Li et al.	A survey of code-based change impact analysis techniques	2013
E83	L. Lima et al.	Test case prioritization based on data reuse an experimental study	2009
E84	C.-T. Lin et al.	History-based test case prioritization with software version awareness	2013
E85	W. Liu et al.	The research of the test case prioritization algorithm for black box testing	2014
E86	J. Lv et al.	On the gain of measuring test case prioritization	2013
E87	Z. Ma, J. Zhao	Test case prioritization based on analysis of program structure	2008
E88	R. Maheswari, D. JeyaMala	A novel approach for test case prioritization	2013
E89	R. Malhotra, D. Tiwari	Development of a framework for test case prioritization using genetic algorithm	2013
E90	A. Malishevsky et al.	Modeling the cost-benefits tradeoffs for regression testing techniques	2002
E91	C. Malz et al.	Prioritization of test cases using software agents and fuzzy logic	2012

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E92	L. Mariani et al.	Compatibility and regression testing of cots-component-based software	2007
E93	L. Mariani et al.	G-ranktest: Regression testing of controller applications	2012
E94	D. Marijan et al.	Test case prioritization for continuous regression testing	2013
E95	W. Masri, M. El-Ghali	Test case filtering and prioritization based on coverage of combinations of program elements	2009
E96	L. Mei et al.	Test case prioritization for regression testing of service-oriented business applications	2009
E97	L. Mei et al.	Xml-manipulating test case prioritization for xml-manipulating services	2011
E98	H. Mei et al.	A static approach to prioritizing junit test cases	2012
E99	S. Mirarab, L. Tahvildari	An empirical study on Bayesian network-based approach for test case prioritization	2008
E100	S. Mirarab et al.	Size-constrained regression test case selection using multicriteria optimization	2012
E101	S. Mohanty et al.	A model based prioritization technique for component based software retesting using uml state chart diagram	2011
E102	S. Mohapatra, S. Prasad	Evolutionary search algorithms for test case prioritization	2013
E103	P. Nagahawatte, H. Do	The effectiveness of regression testing techniques in reducing the occurrence of residual defects	2010
E104	C. Nguyen et al.	Change sensitivity based prioritization for audit testing of webservice compositions	2011
E105	C. Nguyen et al.	Test case prioritization for audit testing of evolving web services using information retrieval techniques	2011
E106	J. Ouriques et al.	On the influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing	2013
E107	C. Panigrahi, R. Mall	Model-based regression test case prioritization	2010
E108	H. Park et al.	Historical value-based approach for costcognizant test case prioritization to improve the effectiveness of regression testing	2008
E109	J. Petke et al.	Efficiency and early fault detection with lower and higher strength combinatorial interaction testing	2013

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E110	M. Praba, D. Mala	Critical component analyzer - a novel test prioritization framework for component based real time systems	2011
E111	N. Prakash, T. Rangaswamy	Modular based multiple test case prioritization	2012
E112	X. Qu et al.	Combinatorial interaction regression testing: A study of test case generation and prioritization	2007
E113	B. Qu et al.	Test case prioritization for black box testing	2007
E114	X. Qu et al.	Configuration-aware regression testing: an empirical study of sampling and prioritization	2008
E115	B. Qu et al.	Test case prioritization for multiple processing queues	2008
E116	X. Qu	Configuration aware prioritization techniques in regression testing	2009
E117	X. Qu, M. Cohen	A study in prioritization for higher strength combinatorial testing	2013
E118	K. Rajarathinam, S. Natarajan	Test suite prioritisation using trace events technique	2013
E119	M. K. Ramanathan et al.	Phalanx: A graph-theoretic framework for test case prioritization	2008
E120	E. Rogstad et al.	Industrial experiences with automated regression testing of a legacy database application	2011
E121	G. Rothermel et al.	Test case prioritization: an empirical study	1999
E122	G. Rothermel et al.	Prioritizing test cases for regression testing	2001
E123	G. Rothermel et al.	On test suite composition and cost-effective regression testing	2004
E124	M. J. Rummel et al.	Towards the prioritization of regression test suites with data flow information	2005
E125	S. Sabharwal et al.	A genetic algorithm based approach for prioritization of test case scenarios in static testing	2011
E126	S. Said et al.	Prioritizing interaction test suite for t-way testing	2011
E127	E. Salecker et al.	Calculating prioritized interaction test sets with constraints using binary decision diagrams	2011
E128	M. Salehie et al.	Prioritizing requirements-based regression test cases: A goal-driven practice	2011

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E129	Y. Salem, R. Hassan	Requirement-based test case generation and prioritization	2010
E130	S. Sampath et al.	Prioritizing user-session-based test cases for web applications testing	2008
E131	S. Sampath et al.	A tool for combinationbased prioritization and reduction of user-session-based test suites	2011
E132	S. Sampath et al.	A uniform representation of hybrid criteria for regression testing	2013
E133	A. B. Sánchez et al.	The drupal framework: A case study to evaluate variability testing techniques	2013
E134	A. Sanchez et al.	A comparison of test case prioritization criteria for software product lines	2014
E135	P. Sapna, H. Mohanty	Prioritizing use cases to aid ordering of scenarios	2009
E136	P. Sapna, H. Mohanty	Prioritization of scenarios based on uml activity diagrams	2009
E137	M. Sherriff et al.	Prioritization of regression tests using singular value decomposition with empirical change records	2007
E138	C. Simons, E. Paraiso	Regression test cases prioritization using failure pursuit sampling	2010
E139	Y. Singh et al.	Test case prioritization using ant colony optimization	2010
E140	A. M. Smith, G. Kapfhammer	An empirical study of incorporating cost into test suite reduction and prioritization	2009
E141	H. Srikanth et al.	System test case prioritization of new and regression test cases	2005
E142	H. Srikanth et al.	Reducing field failures in system configurable software: Cost-based prioritization	2009
E143	H. Srikanth, M. Cohen	Regression testing in software as a service: An industrial case study	2011
E144	H. Srikanth, S. Banerjee	Improving test efficiency through system test prioritization	2012
E145	A. Srivastava, J. Thiagarajan	Effectively prioritizing tests in development environment	2002
E146	P. R. Srivastva et al.	Test case prioritization based on requirements and risk factors	2008
E147	M. Staats et al.	Oracle-centric test case prioritization	2012
E148	W. Sun et al.	Multi-objective test case prioritization for gui applications	2013
E149	S. Sun et al.	Research on optimization scheme of regression testing	2013

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E150	F. Sun, Y. Li	Regression testing prioritization based on model checking for safety-crucial embedded systems	2013
E151	P. Tonella et al.	Using the case-based ranking methodology for test case prioritization	2006
E152	M. Tyagi, S. Malhotra	Test case prioritization using multi objective particle swarm optimizer	2014
E153	K. R. Walcott et al.	Timeaware test suite prioritization	2006
E154	S. Wang et al.	Multi-objective test prioritization in software product line testing: An industrial case study	2014
E155	X. Wang, H. Zeng	Dynamic test case prioritization based on multi-objective	2014
E156	S. Weissleder	Towards impact analysis of test goal prioritization on the efficient execution of automatically generated test suites based on state machines	2011
E157	Z. D. Williams, G. Kapfhammer	Using synthetic test suites to empirically compare search-based and greedy prioritizers	2010
E158	W. Wong et al.	A study of effective regression testing in practice	1997
E159	K. Wu et al.	Test case prioritization incorporating ordered sequence of program elements	2012
E160	H. Wu et al.	Test suite prioritization by switching cost	2014
E161	H. Xian	A test case design algorithm based on priority techniques, in: Management of e-Commerce and e-Government (ICMeCG)	2011
E162	D. Xu, J. Ding	Prioritizing state-based aspect tests	2010
E163	S. Yoo et al.	Fault localization prioritization: Comparing information-theoretic and coverage-based approaches	2013
E164	D. You et al.	An empirical study on the effectiveness of time-aware test case prioritization techniques	2011
E165	Y. Yu et al.	Generating, selecting and prioritizing test cases from specifications with tool support	2003
E166	Y. T. Yu, M. F. Lau	Fault-based test suite prioritization for specification-based testing	2012
E167	T. Yu et al.	Simrt: An automated framework to support regression testing for data races	2014
E168	J. Yuan, Z. Jiang	Constructing prioritized interaction test suite with interaction relationship	2010
E169	K. Zhai et al.	Taking advantage of service selection: A study on the testing of location-based web services through test case prioritization	2010

Tabela A.2: continuação

Estudo	Autores	Título	Ano de publicação
E170	K. Zhai, W. Chan	Point-of-interest aware test case prioritization: Methods and experiments	2010
E171	K. Zhai et al.	Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study	2014
E172	X. Zhang et al.	Test case prioritization based on varying testing requirement priorities and test case costs	2007
E173	L. Zhang et al.	Time-aware test-case prioritization using integer linear programming	2009
E174	L. Zhang et al.	Prioritizing junit test cases in absence of coverage information	2009
E175	X. Zhang, B. Qu	An improved metric for test case prioritization	2011
E176	L. Zhang et al.	Bridging the gap between the total and additional test-case prioritization strategies	2013
E177	Z. Zhi-hua et al.	Test case prioritization for regression testing based on function call path	2012
E178	Z. Q. Zhou et al.	On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites	2012