Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Uma Abordagem Baseada em Métricas de Componentes de Software para a Priorização de Casos de Teste

Dennis Sávio Martins da Silva

Número de Ordem PPGCC: M001

Teresina-PI, Agosto de 2017

Dennis Sávio Martins da Silva

Uma Abordagem Baseada em Métricas de Componentes de Software para a Priorização de Casos de Teste

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Ricardo de Andrade Lira Rabêlo

Coorientador: Pedro de Alcântara dos Santos Neto

Teresina-PI

Agosto de 2017

FICHA CATALOGRÁFICA
Universidade Federal do Piauí
Biblioteca Comunitária Jornalista Carlos Castello Branco
Serviço de Processamento Técnico

S586a Silva, Dennis Sávio Martins da.
Uma abordagem baseada em métricas de componentes de softwares para priorização de casos de teste / Dennis Sávio Martins da Silva. -- 2017.
84 f. : il.

Dissertação (Mestrado) – Universidade Federal do Piauí, Centro de Ciências da Natureza, Programa de Pós-graduação em Ciência da Computação, Teresina, 2017.
"Orientação: Prof. Dr. Ricardo de Andrade Lira Rabêlo."
"Coorientação: Prof. Dr. Pedro de Alcântara dos Santos."

1. Software - Testes. I. Título.

CDD 005.14

Elaborado por Thais Vieira de Sousa Trindade - CRB-3/1282

"Uma Abordagem Baseada em Métricas de Componentes de Software para a Priorização de Casos de Teste"

DENNIS SÁVIO MARTINS DA SILVA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências da Natureza da Universidade Federal do Piauí, como parte integrante dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Aprovada por:

Prof. Ricardo de Andrade Lira Rabêlo

(Presidente da Banca Examinadora)

Prof. Gustavo Augusto Lima de Campos

(Examinador Externo)

Prof. José Valdemir dos Reis Júnior

(Examinador Interno)

Prof. Pedro de Alcântara dos Santos Neto

(Examinador Interno)

Teresina – PI, 15 de setembro de 2017

Aos meus pais Rosineide Martins Santana Silva e Domingos Sávio da Silva, por todo o suporte, apoio e confiança em mim depositados.

Agradecimentos

Assim como tudo na ciência, este trabalho foi resultado da colaboração de inúmeras pessoas. Algumas delas influenciaram de forma mais direta, e puderam ser mencionadas ao longo do texto e na seção de referências. Mas muitas pessoas mais próximas contribuíram para a realização desta dissertação.

Agradeço muito à minha família. A meus pais, Rosineide Martins Santana Silva e Domingos Sávio da Silva, que sempre investiram na minha educação e nunca deixaram que nada faltasse para que eu pudesse ter um bom futuro. E que continuam me ensinando, com base em suas experiências de vida. À minha irmã do meio, contadora e futura advogada Deny Sávia, por toda a ajuda que me ofereceu enquanto na capital. À minha irmã mais nova, Sarah Martins, por toda a alegria e orgulho que traz a nossa família. A todos os tios(as), primos(as) e avós, que sempre estiveram na torcida pelo meu sucesso!

À namorada com quem sempre sonhei, Tabata Cover, pela inspiração, apoio, carinho, compreensão, e por ter me acompanhado durante todas as alegrias e dificuldades que enfrentei. Obrigado por tornar essa caminhada mais suave, essa vitória é nossa! Agradeço também aos seus pais, Cristina Muller e Ilceu Cover, por sempre terem me recebido tão bem em sua família.

A todo o excepcional grupo de docentes do PPGCC, que a cada dia trabalha para projetar nosso estado dentro do cenário acadêmico nacional. Gostaria de mencionar especialmente os professores Vinícius Machado, Rodrigo Veras e Raimundo Moura, com os quais tive oportunidade de ter contato mais direto nas disciplinas. De cada um de vocês, absorvi algo que já emprego na minha atividade docente.

Agradeço a meu professor e orientador, Ricardo Lira. Um homem que aparentemente desconhece o conceito de férias! Muito obrigado por ter acreditado em mim desde o começo e pela oportunidade que me ofereceu. Obrigado pela disponibilidade e por sempre compartilhar sua experiência (acadêmica e de vida) com todos com que o procuram, admiram, e veem em você um exemplo de profissional a seguir.

Ao meu coorientador, Pedro Santos Neto, por todos os conselhos e direcionamentos ao longo de todo o curso, e especialmente durante a realização da pesquisa. As dicas e até mesmo os puxões de orelha foram essenciais para a conclusão deste trabalho.

Ao professor Ricardo Britto, que mesmo fora do Brasil sempre esteve disponível para ajudar no trabalho. Obrigado pelas valiosíssimas orientações na parte de análise estatística dos resultados. O Brasil precisa de você!

A todos os colegas do PPGCC, nas duas turmas em que tive a oportunidade de

participar (como aluno especial e regular). Aos amigos Pedro Almir, Matheus Campanhã e Werney Lira, que participaram da elaboração da abordagem que originou este trabalho. Não posso deixar de agradecer especialmente aos colegas Jailson Leocádio, Denise Alves e Marcos Frasão, com quem tive envolvimento mais direto na realização das atividades do curso e que possuem portanto grande importância para esta vitória.

Além dos amigos que fiz no PPGCC, neste período de estudo tive a oportunidade de conviver um pouco mais com os professores de Sistemas de Informação que também estão se qualificando no PPGCC. Fica o abraço especial aos amigos de jornada Ivenilton Moura, Alan Rafael, Leonardo Sousa, Ismael Leal, Francisco Imperes e Alan Jheyson, pelas histórias, risadas, asfalto, poeira e gasolina compartilhada nesses anos!

Agradeço aos colegas professores do curso de Sistemas de Informação, do Campus Senador Helvídio Nunes de Barros em Picos-PI. Grupo bastante coeso e solidário. Minha gratidão especial às três chefas, Patrícia Medyna, Alcilene Dalília e Patrícia Vieira, que sempre fizeram o possível para conciliar nossos horários de trabalho e qualificação.

Ainda no curso de Sistemas de Informação, sou imensamente grato aos meus orientandos e alunos, que foram bastante proativos ao desempenhar suas atividades nos momentos em que eu estive mais sobrecarregado, e apresentaram ótimos resultados!

A todos os participantes do Laboratório OASIS, equipe super dedicada, prestativa e unida que sem dúvida ainda alcançará grande visibilidade. Abraço especial ao Vinícius Nunes, que está realizando um trabalho relacionado a esta dissertação.

Por fim, a todos os meus amigos. Não é justo citar um subgrupo e acabar esquecendo alguém, mas deixo um abraço especial a Ivaldir Honório e Ariadnes Rodrigues, com quem tive meus primeiros passos na pesquisa e pude aprender bastante antes de iniciar a pesquisa. E não posso deixar de citar o *good blood headbanger* Renan Vieira, provavelmente a única pessoa com quem conseguirei falar de pesquisa fazendo referências a Hermes e Renato.

"A coisa mais essencial do espírito vivo de um homem é sua paixão pela aventura. A alegria da vida vem de nossos encontros com novas experiências."

(Christopher McCandless)

Resumo

O teste de software é a principal forma de detectar falhas e avaliar a qualidade de um programa. Dentre os tipos de teste de software, podemos destacar o teste de regressão, que reexecuta os casos de teste do sistema para verificar se modificações em uma versão do programa não trouxeram prejuízo ao funcionamento de módulos que estavam atuando corretamente em versões anteriores. A forma mais comum de teste de regressão é a reexecução de todos os casos de teste do sistema, mas o crescimento no porte de um sistema de software pode dificultar essa abordagem devido a restrições de tempo ou de custos. Dentre as abordagens para resolver esse problema e melhorar o teste de regressão, podemos destacar a priorização de casos de teste, que consiste na alteração da sequência de execução dos casos de teste, buscando maximizar alguma propriedade da *suite* (conjunto) de testes. Este trabalho tem por objetivo priorizar casos de teste, buscando maximizar a taxa de detecção de falhas do teste de regressão. Para isso, é proposta uma abordagem em três etapas: na primeira etapa, são atribuídos valores de criticidade aos componentes do software por meio de um sistema de inferência *fuzzy*, cujas entradas são métricas de software relacionadas à susceptibilidade a falhas; na segunda etapa, é calculada a criticidade dos casos de teste, com base na criticidade dos componentes de software por eles cobertos; na terceira etapa acontece a priorização dos casos de teste, que emprega otimização por colônia de formigas para ordenar os casos de teste considerando suas criticidades, tempos de execução e histórico de detecção de falhas. As soluções foram avaliadas com base nas métricas APFD (*Average Percentage of Fault Detection*, que representa a taxa de detecção de falhas) e APFD_C (variação do APFD que considera os custos de execução dos casos de teste e a severidade das falhas). Foram realizados experimentos com programas em linguagem C obtidos no repositório SIR (Software-artifact Infrastructure Repository), e os resultados obtidos com a abordagem foram comparados à não-ordenação das *suites*, à ordenação obtida por meio de uma busca exaustiva e a uma ordenação obtida por um algoritmo guloso que considerou a criticidade, histórico de detecção de falhas e tempo de execução dos casos de teste. Além disso, os resultados foram submetidos a um teste de sanidade e comparados à ordenação aleatória, apresentando fortes indícios de adequação ao problema.

Palavras-chaves: Priorização de testes. Teste de Regressão. Sistema de Inferência *Fuzzy*. Otimização por Colônia de Formigas.

Abstract

Software testing is the main way of detecting faults and evaluating the quality of a program. Among the types of software testing, we can highlight the regression testing, which reexecutes the system's test cases to check if changes in a version of the program did not bring prejudice to the functioning of modules which were working properly in previous versions. The most common way of regression testing is the reexecution of all the system's test cases, but the growth of the system's size can hamper this approach due to time and cost restrictions. Among the approaches to solve this problem and improve the regression testing, we can highlight the test case prioritization, that consists in changing the execution sequence of the test cases, aiming to maximize some property of the test suite. This work aims to prioritize test cases, seeking to maximize the rate of fault detection. In this regard, a three-step approach is proposed: at the first step, there are attributed criticality values to the software components through a fuzzy inference system, whose inputs are software metrics related to fault proneness; at the second step, the test cases criticality is calculated, based on the criticality of the software components covered by them; at the third step occurs the test prioritization, which uses ant colony optimization to order the test cases considering their criticality, execution times and history of faults. The solutions were evaluated based on the APFD (*Average Percentage of Fault Detection*) and APFD_C (APFD variation which coniders the cost of the test cases and the severity of the faults) metrics. There were performed experiments with C programs from the repository SIR (Software-artifact Infrastructure Repository), and the results obtained with the approach were compared to the non-ordination of the test suites, to the ordination obtained through an exhaustive search, and to the ordination obtained through a greedy algorithm which considered the criticality, history of fault detection and execution time of the test cases. Furthermore, the results were submitted to a sanity check and compared to a random search, showing strong evidence of suitability for the problem.

Keywords: Test Prioritization. Regression Testing. Fuzzy Inference System. Ant Colony Optimization.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACO	*Ant Colony Optimization*
APBC	*Average Percentage Block Coverage*
APDC	*Average Percentage Decision Coverage*
APFD	*Average Percentage of Fault Detection*
$APFD_C$	*Average Percentage of Fault Detection - Costs*
APSC	*Average Percentage Statement Coverage*
BCV	*Business Criticality Value*
CC	*Criticidade da Classe*
DUA	*Definition-use Association*
FC	*Criticidade do Componente de Software*
MMAS	*Max-Min Ant System*
NSGA-II	*Non-dominated Sorting Generic Algorithm - II*
RTS	*Regression Test Selection*
SIR	*Software-Artifact Infrastructure Repository*
SQFD	*Software Quality Function Deployment*
TC	*Criticidade do Caso de Teste*
TSM	*Test Suite Minimization*
TSR	*Test Suite Reduction*
TCS	*Test Case Selection*

Lista de símbolos

$\in$	Pertence a
$\neq$	Diferente de
$\forall$	Para todo
$\geq$	Maior que ou igual
μ	Letra grega Miu
Σ	Letra grega Sigma
τ	Letra grega Tau
η	Letra grega Eta
α	Letra grega Alfa
β	Letra grega Beta
Δ	Letra grega Delta

Sumário

Introdução

Contexto e Motivação

Em 2016, foi reportado que, em diversas áreas da indústria, pelo menos 4.4 bilhões de pessoas e 1.1 trilhões em ativos foram impactados por falhas de software (TRICENTIS, 2017). Ao passo em que a tecnologia evolui e inova, aumenta também a necessidade de proteger os usuários das consequências do mau funcionamento de programas. Além disso, com o aumento da demanda e dependência por sistemas de software nos mais diversos setores sociais e econômicos, existe a necessidade de meios para garantir a qualidade dos produtos em desenvolvimento. Para isso, além da adoção de boas metodologias para o projeto pode-se destacar a atividade de teste de software.

Teste de software é a verificação da adequação de um programa aos requisitos e funcionalidades previstos durante o projeto (PRESSMAN, 2015; BOURQUE; FAIRLEY, 2014). A finalidade mais evidente do teste de software é permitir que a equipe de desenvolvimento encontre falhas antes que os usuários finais o façam, mas o teste não se restringe a isso. Mais do que uma mera detecção de falhas, o teste permite aos desenvolvedores avaliar a qualidade do produto que está sendo desenvolvido.

A atividade de teste consiste na execução de um conjunto finito de casos de teste (ABRAN et al., 2001). Um caso de teste consiste da combinação de um conjunto de entradas, uma função de execução e um conjunto de saídas esperadas (ASKARUNISA; SHANMUGAPRIYA; RAMARAJ, 2010). Casos de teste podem ser projetados com diversos objetivos, como: verificar a implementação de especificações funcionais; mensurar a confiabilidade do sistema; avaliação de usabilidade; aferição de desempenho e comportamento do sistema sob stress; avaliar a aceitação do software; dentre outros. Ao conjunto de casos de teste do sistema denomina-se *suite* de testes. Dependendo do aspecto do sistema que se deseja avaliar podem ser aplicados diversos tipos de teste de software, como testes de integração, testes funcionais, testes de aceitação, testes de segurança, dentre outros (PRESSMAN, 2015; SOMMERVILLE, 2015).

Por mais bem sucedido que seja, o desenvolvimento de um software envolve constante evolução e manutenção do produto por diversos motivos, como aumento no número de requisitos, adição de novas funcionalidades, correção de *bugs*, melhoria na manutenibilidade e eficiência, etc. Porém, as atualizações realizadas em um sistema trazem consigo o risco de afetar de maneira adversa a funcionalidade do produto.

Para reduzir esse risco, pode-se destacar dentre os tipos de teste de software o teste de regressão, que consiste em reexecutar os casos de teste do sistema, visando verificar

se modificações não inseriram falhas ou efeitos colaterais inesperados em módulos que estavam funcionando adequadamente em versões anteriores do projeto (IEEE, 1990).

Definição do Problema

A abordagem mais comum para o teste de regressão é a reexecução de todos os casos de teste do sistema (*retest-all*) (YOO; HARMAN, 2010). Entretanto, conforme dá-se o crescimento da quantidade de módulos de um sistema de software, tende a ocorrer também um incremento no número de casos de teste necessários para a verificação satisfatória do sistema. O teste de regressão pode ser uma atividade onerosa, podendo consumir cerca de 80% do orçamento destinado ao teste do sistema (LU et al., 2016). Há relatos de projetos onde a execução do conjunto completo de testes chega a consumir semanas (ROTHERMEL et al., 1999) ou até meses (ELBAUM et al., 2003). E mesmo que a duração do teste de regressão seja curta, o custo ainda deve ser considerado, devido aos recursos humanos envolvidos na preparação, execução e monitoramento da atividade.

Todos esses fatores podem inviabilizar o reteste completo do sistema e evidenciam a necessidade de abordagens para lidar com os custos envolvidos no teste de regressão na ocorrência de limitações, sendo elas temporais, relacionadas a recursos humanos, ou de outras naturezas. Além disso, quanto mais prematura for a detecção das falhas mais rapidamente a correção do sistema poderá ser iniciada, então é importante que independente da ocorrência de limitações temporais busque-se também um melhor aproveitamento do tempo destinado ao teste de regressão.

Os trabalhos relacionados à melhoria do teste de regressão costumam considerar o custo (geralmente o tempo necessário à execução dos casos de teste) e a eficácia (taxa com que o código é coberto ou que as falhas são detectadas) do conjunto de casos de teste (YOO; HARMAN, 2010). As abordagens propostas para a melhoria do custo-eficácia do teste de regressão costumam ser baseadas em diversas informações, como o histórico de falhas, a cobertura do código, o tempo de execução dos testes, o valor de criticidade do teste para o negócio, dentre outras (CATAL; MISHRA, 2013).

As propostas para a melhoria do custo-eficácia do teste de regressão costumam ser categorizadas nas seguintes abordagens (CATAL; MISHRA, 2013; YOO; HARMAN, 2007):

- Seleção de casos de teste: escolha de um subconjunto dos casos de teste de um programa considerando algum critério de interesse, como a cobertura de código, o histórico de modificações do programa, dentre outros (ROTHERMEL; HARROLD, 1996);
- Minimização de casos de teste: remoção permanente de casos de teste considerados

redundantes (ROTHERMEL et al., 2002), ou seja, que cobrem apenas parte dos requisitos satisfeitos por outro caso de teste;

- Priorização de casos de teste: alteração da sequência de execução dos casos de teste, buscando uma ordenação que maximize propriedades da *suite* de testes relacionadas à ordem de execução, como a taxa de detecção de falhas e a taxa de cobertura.

Seguindo por estratégias diferentes, o objetivo das três abordagens é o mesmo: fazer com que o tempo disponível para o teste de regressão seja melhor aproveitado com base em algum critério de avaliação. As abordagens de seleção e minimização lidam com a questão da disponibilidade de tempo para o teste de regressão, mas podem aumentar o custo do software ao omitir a execução de casos de teste importantes que detectam tipos específicos de falhas (DO et al., 2010). Já as abordagens baseadas em priorização mostram-se interessantes por não envolver nenhum tipo de redução da *suite* de testes, e sim a busca por uma sequência de execução para os casos de teste que, em caso de interrupção prematura, permita ao testador obter maiores benefícios da verificação (YOO; HARMAN, 2010).

Com isso, o problema a que este trabalho se propõe solucionar apresenta a seguinte definição formal: dado um conjunto de casos de testes, encontrar uma ordenação que permita maximizar a detecção de falhas do teste de regressão considerando informações disponíveis no teste de regressão, como o histórico de detecção de falhas, além de métricas obtidas por meio da análise do código e relacionadas à susceptibilidade a falhas dos componentes do código.

Visão Geral da Proposta

Este trabalho apresenta uma abordagem em três etapas que utiliza informações disponíveis durante a realização do teste de regressão para a priorização de casos de teste, visando o aumento da taxa de detecção de falhas do conjunto de testes.

Na primeira etapa, é atribuído um valor de criticidade a cada componente do software. Como não existe métrica precisa para definir a criticidade (importância de execução) de um caso de teste, definiu-se na abordagem que esse valor seria atribuído com base na criticidade dos componentes por ele cobertos. O grau de criticidade de um componente reflete sua susceptibilidade a falhas, e foi obtido por meio de um sistema de inferência *fuzzy* (ZADEH, 1996) cujas entradas (complexidade ciclomática, o número de caminhos de execução dentro de um trecho de código; e instabilidade, o nível de inter-relação entre componentes do software (MCCABE, 1976; MARTIN, 2003)) são métricas de software relacionadas à susceptibilidade a falhas.

Na segunda etapa é calculada a criticidade dos casos de teste. Esse valor reflete o potencial de detecção de falhas pelos casos de teste, e é obtido com base na criticidade dos componentes de software que ele cobre. É importante notar que a cobertura dos casos de teste é um critério bastante empregado em abordagens para o teste de regressão, e a abordagem proposta considera tanto a cobertura (na forma da quantidade de componentes do software cobertos) quanto o peso dos componentes cobertos (na forma da criticidade dos componentes).

Na terceira etapa acontece a priorização dos casos de teste, e os casos de teste são ordenados considerando suas criticidades, tempos de execução e históricos de detecção de falhas. O problema é formulado como o do caixeiro viajante, motivo pelo qual é utilizada a otimização por colônia de formigas.

As métricas empregadas para mensurar a qualidade das soluções foram o APFD (*Average Percentage of Fault Detection*, que representa a taxa de detecção de falhas (ELBAUM; MALISHEVSKY; ROTHERMEL, 2000)) e o APFD_C (variação do APFD que considera o custo (tempo de execução) dos casos de teste e a severidade das falhas (ASKARUNISA; SHANMUGAPRIYA; RAMARAJ, 2010)). Essas métricas consideram o histórico de detecção de falhas dos casos de teste, considerando que casos de teste que já encontraram falhas possuem poder de detecção comprovado e que essa informação é de obtenção possível durante a realização do teste de regressão. Os objetos empregados na experimentação são programas obtidos em um repositório aberto de objetos de software (DO; ELBAUM; ROTHERMEL, 2005) que possuem conjuntos de casos de teste e de falhas semeadas.

Os resultados obtidos com a abordagem foram comparados à ordenação original das *suites*; à ordenação ótima, obtida por meio de uma busca exaustiva, quando possível; e a uma ordenação obtida por um algoritmo guloso. Além disso, os resultados foram submetidos a um teste de sanidade, que consiste na comparação com os resultados de uma ordenação aleatória. A comparação se deu em termos de significância estatística e prática, e seus resultados apresentaram fortes indícios de adequação ao problema.

Objetivos

O objetivo principal desse trabalho é propôr uma alternativa para a melhoria da taxa de detecção de falhas no teste de regressão. Tal alternativa consiste de uma abordagem de priorização de casos de teste, baseada na proposta em (SILVA et al., 2016). Para alcançar esse objetivo, foi necessária a execução de alguns objetivos específicos:

- Apresentação dos principais paradigmas para melhoria do custo-eficácia do teste de regressão (seleção, minimização e priorização de casos de teste);

- Leitura de trabalhos relacionados, para identificação de pesquisas recentes na área de pesquisa. Isso permitiu melhor compreender o funcionamento e identificar as lacunas das principais propostas relacionadas à melhoria do teste de regressão;

- Implementação da abordagem definida e realização da instrumentação (análise do código) dos objetos empregados na experimentação para obtenção de métricas e informações necessárias à execução dos experimentos;

- Avaliação da abordagem por meio de um estudo experimental empregando programas oriundos de um repositório público de objetos de software para experimentação em Engenharia de Software.

Justificativa

Existem na literatura trabalhos que priorizam casos de teste com base em seu potencial para a detecção de falhas (ROTHERMEL et al., 2001). Apesar de existirem trabalhos que consideram o grau de criticidade dos testes para o negócio (ACHARYA; KHANDAI; MOHAPATRA, 2012; KHANDAI; ACHARYA; MOHAPATRA, 2011), não foram encontrados trabalhos que atribuíssem criticidade aos casos de teste com base nas características dos componentes por eles cobertos. No entanto, pode ser interessante dar maior ênfase à correção de funcionalidades do programa tidas como mais importantes, ou mais susceptíveis a falhas. Com base nessa lacuna, a abordagem proposta neste trabalho atribui níveis de criticidade (importância para o teste) aos componentes do software considerando duas métricas associadas à ocorrência de falhas (GILL; SIKKA, 2011): a complexidade ciclomática e a instabilidade.

Existe uma grande quantidade de trabalhos sobre priorização de casos de teste que empregam a cobertura como critério de avaliação. Entretanto, os trabalhos costumam se ater à taxa com que o código é coberto, sem se preocupar com a importância do código que está sendo coberto. A abordagem proposta neste trabalho emprega a criticidade dos casos de teste, considerando a cobertura dos casos de teste, ao mesmo passo em que assimila a criticidade dos componentes de software que eles cobrem.

Um indicativo de qualidade bastante empregado na avaliação de conjuntos de casos de teste é a taxa de detecção de falhas. Essa informação é quantificada pela métrica APFD, que representa a taxa em que as falhas (presentes uma versão anterior) teriam sido detectadas se os casos de teste tivessem sido executados em determinada ordem. Essa métrica não garante que de fato o teste irá encontrar falhas mais rapidamente, visto que é baseada na ocorrência de falhas passadas e a localização das falhas na versão sob teste só estaria disponível ao final do teste. Entretanto, o histórico de detecção de falhas oferece evidências da eficácia de um caso de teste, e é uma informação empregada em alguns

trabalhos de priorização. Por esse motivo, a abordagem proposta neste trabalho avalia a qualidade das soluções obtidas com base em suas taxas de detecção de falhas.

A abordagem proposta busca, então, oferecer uma alternativa para a priorização de casos de teste de regressão que parte de entradas presentes em um ambiente de teste de regressão (por meio da análise de código ou do histórico de execução e detecção de falhas) e entrega uma ordenação de casos de teste que visa maximizar a taxa de detecção de falhas, considerando não somente o histórico de detecção de falhas e a cobertura do código pelos casos de teste, mas também o potencial de ocorrência de falhas dos componentes do software.

Contribuições

A principal contribuição desse trabalho é a priorização de casos de testes visando a maximização da taxa de detecção de falhas de *suites* de teste, por meio de uma abordagem que considera métricas de software relacionadas à ocorrência de falhas, assim como o tempo de execução e o histórico de detecção de falhas dos casos de teste. Além dessa contribuição principal, podemos elencar outras contribuições:

- Uma atualização da abordagem de priorização e seleção de casos de teste proposta em (SILVA et al., 2016), buscando sanar limitações e apresentar melhorias como: a remoção da técnica de seleção de casos de teste visando melhor explorar o potencial de detecção de falhas da *suite* de testes e os benefícios da técnica de priorização; e a utilização de uma métrica de taxa de detecção de falhas na avaliação das soluções propostas, em substituição à reordenação dos casos de teste em ordem decrescente de criticidade.

- Uma validação da abordagem empregando programas oriundos de um repositório público (DO; ELBAUM; ROTHERMEL, 2005) amplamente utilizado em estudos relacionados a testes de software.

- Uma comparação - em termos de significância estatística e prática - dos resultados alcançados com a abordagem proposta com os obtidos em uma abordagem aleatória.

- Um posicionamento dos resultados da abordagem em relação à ordenação original da *suite* de testes, a uma ordenação obtida com um algoritmo guloso e à melhor ordem possível para a *suite* de testes, obtida por meio de uma busca exaustiva (quando praticável).

Estrutura do Trabalho

O restante deste trabalho está estruturado da seguinte forma: o Capítulo 1 apresenta conceitos relacionados ao teste de regressão, aos sistemas de inferência *fuzzy* e à otimização por colônia de formigas. O Capítulo 2 apresenta os trabalhos relacionados. O Capítulo 3 ilustra a abordagem sugerida em (SILVA et al., 2016) para a priorização e seleção de casos de teste. O Capítulo 4 detalha a abordagem proposta no trabalho. O Capitulo 5 compreende a experimentação realizada e os resultados obtidos, e finalmente o Capítulo 6 apresenta as conclusões, limitações e trabalhos futuros propostos para a continuidade da pesquisa.

1 Referencial Teórico

Neste capítulo são apresentados conceitos ligados ao teste de regressão, bem como os algoritmos empregados no desenvolvimento da abordagem: Os Sistemas *Fuzzy* e Otimização por Colônia de Formigas.

1.1 Teste de Regressão

Um software passa por constante evolução ao longo de seu ciclo de vida, devido a mudanças nos requisitos dos usuários, ao acréscimo de funcionalidades, à correção de *bugs*, à busca por maior eficiência, dentre outras situações. Hetzel e Hetzel (1991) apontam, em um estudo, que a probabilidade de introduzir falhas em um software durante a manutenção oscila entre 50 e 80%. Quando uma nova versão do software está prestes a ser lançada, é necessário verificar se as modificações não afetam de forma inesperada o funcionamento dos componentes do software já existentes e que não foram modificados, acrescentando falhas ou comportamentos inesperados ao sistema. Esse tipo de teste, que considera as alterações do software, é chamado teste de regressão (IEEE, 1990)(YOO; HARMAN, 2010).

Seja P um programa que foi modificado e originou uma versão P_0 e seja T um conjunto de casos de teste desenvolvido para P. O teste de regressão busca validar P_0 de forma a investigar se ele regrediu, *i.e*, se o comportamento que foi previamente verificado como válido em P mostrou-se falho em P_0 (DO et al., 2010).

A abordagem mais simples para o teste de regressão é a *retest-all*, que consiste em executar todos os casos de teste de T em P_0. Com o crescimento do sistema, que se reflete no aumento do número de casos de teste, essa abordagem pode consumir demasiados tempo e recursos a ponto de tornar-se inviável. O teste de regressão é uma atividade onerosa e responsável por parte significativa dos custos com manutenção do software (BEIZER, 1984). Há relatos (ROTHERMEL et al., 1999; ELBAUM et al., 2003) de sistemas cujos conjuntos de teste necessitam de uma grande quantidade de tempo para ser completamente executados, mas nem sempre o tempo disponível para a atividade de teste é suficiente.

Isso evidencia a necessidade de técnicas para melhorar a relação custo-eficácia do teste de regressão, permitindo obter resultados mais cedo e explorar melhor o potencial dos casos de teste em caso de interrupção prematura do teste de regressão. Deve-se lidar com as possíveis restrições, de maneira a permitir a redução dos recursos necessários à execução dos testes, mas o resultado do processo - a *suite* de testes limitada e/ou reordenada - deve sofrer o mínimo prejuízo, especialmente em relação à capacidade de detecção de falhas.

Diversas técnicas já foram propostas no intuito de melhorar o custo-eficácia do

teste de regressão. Em geral, essas técnicas podem ser classificadas em três abordagens básicas: minimização de casos de teste; seleção de casos de teste; e priorização de casos de teste.

Minimização de Casos de Teste

A Minimização de Casos de Teste (*Test Suite Minimization - TSM* ou Redução de Casos de Teste *Test Suite Reduction - TSR*) (ROTHERMEL et al., 2002) consiste na remoção permanente de casos de teste considerados redundantes no que diz respeito à cobertura de determinado requisito. A Figura 1 demonstra uma *suite* de testes que, em decorrência da minimização, quatro dos dez casos de teste da *suite* foram descartados.

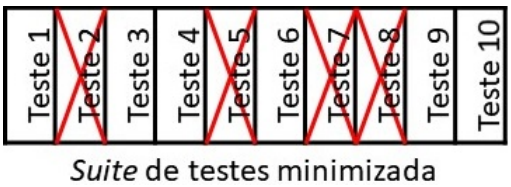

Figura 1 – Minimização de casos de teste.

O problema da minimização de casos de teste pode ser definido como:

Dados: uma *suite* de casos de teste T, um conjunto de requisitos de teste $r_1, ..., r_n$, que podem ser satisfeitos para prover o teste "adequado"do programa, e subconjuntos $T_1, ..., T_n$ de T, associados aos r_is de tal forma que qualquer um dos casos de teste t_j pertencentes a um subconjunto T_i pode cumprir o requisito r_i. *Problema*: encontrar um conjunto representativo T' de casos de teste de T, que satisfaça todos os r_is.

Um requisito de teste r_i é considerado coberto se pelo menos um dos casos de teste t_j em T_i o satisfizer. Para melhorar o efeito da minimização, é interessante que T' seja o conjunto mínimo afetado (*Minimal Hitting Set*) dos T_is. Se um requisito é satisfeito por um único caso de teste, tem-se que este é um caso de teste essencial. Um caso de teste que satisfaz apenas um subconjunto do conjunto de requisitos satisfeito por outro caso de teste é considerado redundante.

Seleção de Casos de Teste

As técnicas baseadas em seleção de casos de teste (*Test Case Selection - TCS*), também conhecida como seleção de teste de regressão (*Regression Test Selection - RTS*) (Figura 2) consistem na seleção de um subconjunto dos casos de teste de um programa, considerando algum critério de interesse (NARCISO; DELAMARO; NUNES, 2014). Rothermel e Harrold (1996) definem o problema da seleção de casos de teste como:

Dados: um programa P, uma versão modificada de P, P', e uma *suite* de testes T. *Problema*: encontrar um subconjunto de T, T', com o qual se testará P'.

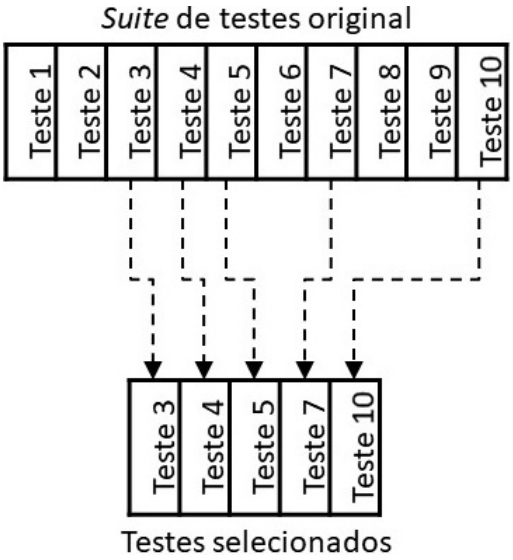

Figura 2 – Seleção de casos de teste.

A seleção pode ser realizada com base em diversos fatores, como a cobertura de código, as modificações do programa, o custo de execução dos testes e o histórico de detecção de falhas (PANICHELLA et al., 2015). Geralmente, a *suite* selecionada é bastante adequada a uma determinada versão do software, por ser criada com base na relação entre os testes e as modificações inseridas nesta versão.

A obtenção de uma seleção ótima (ou seja, que garanta que exatamente os casos de teste que revelam falhas serão escolhidos) pode não ser praticamente possível (KIM; PORTER, 2002), então o ideal é que as técnicas de seleção assegurem que a *suite* de testes selecionada possua o mínimo possível de prejuízo na capacidade de detecção de falhas, quando comparada à *suite* de testes completa. As técnicas de seleção que apresentam esta propriedade são chamadas de técnicas seguras de seleção (ACHARYA; KHANDAI; MOHAPATRA, 2012), no entanto é muito difícil atestar a segurança de uma técnica de seleção, visto que a real capacidade de detecção de falhas de um conjunto de casos de teste só é conhecida após sua execução, e podem existir falhas mapeadas somente por casos de teste não selecionados.

Priorização de casos de teste

As técnicas baseadas em priorização de casos de teste (Figura 3) buscam por uma ordem para a execução dos casos de teste que maximize propriedades como a taxa de detecção de falhas ou a taxa de cobertura do código (CATAL; MISHRA, 2013), permitindo que os objetivos do teste sejam alcançados mais cedo e o potencial dos casos de teste seja

melhor explorado, em caso de interrupção prematura do teste de regressão. A primeira menção à aplicação de priorização vem de (WONG et al., 1995).

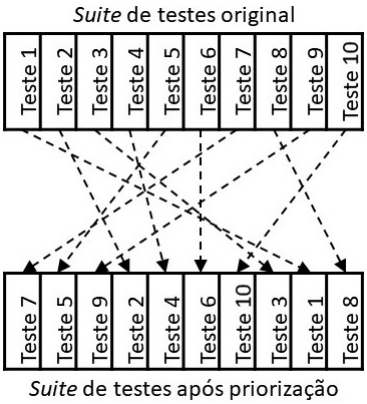

Figura 3 – Priorização de casos de teste

O problema da priorização de casos de teste (ELBAUM; MALISHEVSKY; ROTHERMEL, 2000) pode ser definido como:

Dados: T, o conjunto de casos de teste de um sistema; PT, o conjunto de possíveis permutações de T; f, uma função de PT que quantifique as ordenações em números reais.

Problema: encontrar $T' \in PT$, tal que $(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

Na priorização, o objetivo não é remover testes do conjunto original de testes, ou mesmo deixar testes sem ser executados, mas apenas reordenar o conjunto original de testes.

Considerações sobre a priorização, minimização e seleção de casos de teste

Devido à remoção permanente dos casos de teste, as técnicas de minimização podem reduzir a eficácia da *suite* de testes (PANICHELLA et al., 2015). Cabe enfatizar que, ao contrário das técnicas de minimização, a redução da *suite* de testes promovida pela seleção de casos de teste é apenas a nível de execução. Os casos de teste não-selecionados não são removidos definitivamente do conjunto de casos de teste do sistema, podendo ser selecionados em uma versão posterior (YOO; HARMAN, 2010).

As técnicas baseadas em minimização ou seleção de casos de teste buscam reduzir o tempo de teste. Porém, o fato de suprimirem a execução de casos de teste pode acarretar um aumento no custo do software decorrente da não-detecção de certos tipos de falhas (DO et al., 2010). As técnicas de priorização empregam o conjunto completo de testes, e podem reduzir o custo do teste por meio da paralelização das atividades de *debug* e teste. A priorização melhora o custo-eficácia do teste de regressão, possibilitando uma detecção mais prematura das falhas e um *feedback* mais rápido aos testadores (CATAL; MISHRA, 2013).

1.2 Otimização por Colônia de Formigas

A otimização por colônia de formigas (do inglês *Ant Colony Optimization*, ou ACO) é um modelo de otimização computacional inspirado no comportamento forrageiro de formigas reais (ENGELBRECHT, 2007). Mais especificamente, na capacidade das formigas de encontrar o menor caminho do ninho até uma fonte de comida utilizando feromônio.

Feromônio é uma substância química que as formigas reais distribuem enquanto caminham em busca de alimentos, e que orientam as companheiras que seguirão o caminho posteriormente. Os caminhos com maior tráfego de formigas apresentam maior concentração de feromônio e possuem portanto maior probabilidade de serem escolhidos. Então, o feromônio possui um papel importante ao guiar o processo de decisão das formigas reais. Quando uma formiga escolhe um caminho específico, ela deixa nele o seu próprio feromônio, reforçando a concentração total da substância. Esse comportamento inspira o algoritmo de otimização, onde as formigas são agentes computacionais cujas rotas representam soluções de um problema. O algoritmo ACO envolve dois procedimentos básicos:

- Construção da solução: m formigas constroem em paralelo m soluções para o problema;
- Atualização da concentração de feromônios: as soluções construídas pelas formigas têm sua qualidade mensurada por meio de uma função de avaliação. A atualização da concentração de feromônios é baseada na função de avaliação. Assim, melhores soluções recebem um maior depósito de feromônios.

Além do uso da informação dos feromônios, uma função heurística é utilizada para aprimorar a construção de soluções para o problema. Apesar dos algoritmos ACO serem capazes de solucionar problemas sem utilizar uma função heurística (DORIGO; STüTZLE, 2001), a incorporação de informação heurística normalmente resulta em melhores soluções (DORIGO; BONABEAU; THERAULAZ, 2000).

1.3 Sistemas de Inferência *Fuzzy*

A lógica *fuzzy* é adequada a situações nas quais a informação seja imprecisa ou incerta, e palavras sejam melhores para representar conceitos do que números (ZADEH, 1996; MCNEILL; THRO, 1994). Um sistema de inferência *fuzzy* lida com informação qualitativa. Isto facilita o mapeamento da experiência dos especialistas do domínio de interesse, uma característica bastante aplicável a estimativas e tomada de decisões. As entradas e saídas do sistema são compreensíveis de um ponto de vista linguístico.

Sistemas de inferência *fuzzy* são baseados em regras linguísticas de produção no formato *"Se ... Então..."*. A teoria dos conjuntos *fuzzy* (ZADEH, 1965) e a lógica *fuzzy*

(ZADEH, 1996) provêem a base matemática para lidar com informação imprecisa, incerta ou qualitativa. A lógica *fuzzy* é uma generalização da lógica clássica (*crisp*) capaz de processar elementos em conjuntos *fuzzy*. Conjuntos *fuzzy* são uma generalização dos conjuntos clássicos, cujos elementos podem possuir graus pertinência (μ) apenas parciais. Enquanto a lógica tradicional trabalha com os valores 0 e 1 (falso e verdadeiro) representando falsidade ou verdade absoluta, a lógica *fuzzy* lida com valores contínuos no intervalo [0,1], de forma a lidar com verdades parciais.

Os sistemas de inferência *fuzzy* são constituídos das seguintes partes (Figura 4):

- Interface de entrada (fuzzificação): recebe as variáveis de entrada do sistema, e realiza o processo de conversão dos valores *crisp* de entrada em conjuntos *fuzzy* definidos no universo de cada variável. Os conjuntos são representados por termos linguísticos, como "muito", "pouco", etc.

- Sistema de inferência *fuzzy*: Realiza o raciocínio sobre as regras do sistema e os fatos alimentados na interface de entrada para obtenção de saídas.

- Base de conhecimento: Composta de uma base de regras, que armazena um conjunto de regras *fuzzy* para mapeamento das relações entre valores de entrada e de saída; e uma base de dados, que armazena as funções de pertinência e os universos das variáveis;

- Interface de saída (defuzzificação): o resultado obtido em um sistema de inferência *fuzzy* é um novo conjunto *fuzzy*. Para que esse retorno seja compreendido pelo especialista, é necessário que ele passe por um processo de extração do resultado *crisp* mais representativo do conjunto *fuzzy* resposta.

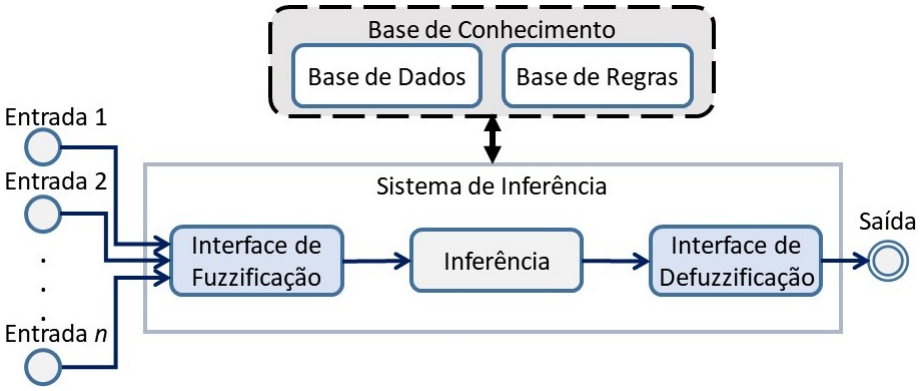

Figura 4 – Sistema de inferência *fuzzy*.

O funcionamento de um sistema *fuzzy* se divide em três etapas:

- Etapa 1: Fuzzificação: mapeamento das entradas numéricas em conjuntos *fuzzy*.

- Etapa 2: Inferência: produção do valor de saída do sistema *fuzzy*, com base nos valores de entrada.
- Etapa 3: Defuzzificação: associação de um valor numérico à saída da etapa de inferência.

2 Trabalhos Relacionados

Para a elaboração deste trabalho, foram analisadas diversas referências no intuito de conhecer as abordagens da comunidade científica para o problema do teste de regressão. Este capítulo apresenta alguns dos trabalhos que nortearam a elaboração desta dissertação.

Yoo e Harman (2010) investigam, por meio de um *survey*, o estado-da-arte em priorização, seleção e minimização de casos de teste. O estudo apresenta as principais alternativas apresentadas na literatura para cada uma das três classes de técnicas para otimização do teste de regressão. No trabalho também são discutidos problemas em aberto, e oportunidades de pesquisa no assunto.

Catal e Mishra (2013) propõem um Mapeamento Sistemático de Literatura visando investigar técnicas, aspectos e tendências relacionados à priorização de casos de teste. Observou-se que uma quantidade de trabalhos empregava técnicas baseada em modelos ou gulosas. No que diz respeito à avaliação das soluções, a métrica dominante nos trabalhos foi a taxa de detecção de falhas (APFD - *Average Percentage of Faults Detected*). Com base no levantamento, os autores deram alguns direcionamentos a pesquisadores da área, como: Dar preferência ao uso de *datasets* públicos ou de projetos reais da industria na experimentação e, ao propor novas técnicas de priorização, realizar uma análise da eficácia dos resultados com base em métricas conhecidas de avaliação.

O mapeamento constatou ainda uma aplicação significativa de métodos baseados em cobertura de código. A ideia por trás disso é a de que, uma vez que para que uma falha em um componente estrutural seja detectada esse componente precisa ser executado, o adiantamento da cobertura estrutural aumenta a chance de adiantar também a cobertura das falhas pelos casos de teste (HAO et al., 2016). Ou seja: Mesmo que a ordenação promovida pela priorização não influencie a quantidade de falhas que seriam encontradas no teste de regressão, espera-se que quanto mais rápido o código for coberto pelos casos de teste, maiores as chances de encontrar as falhas de forma mais precoce durante a execução dos testes. Yoo e Harman (2010) mencionam diversas propostas na literatura baseadas em cobertura, destacando (ELBAUM; MALISHEVSKY; ROTHERMEL, 2000; ELBAUM; GABLE; ROTHERMEL, 2001; ELBAUM; MALISHEVSKY; ROTHERMEL, 2001; ELBAUM; MALISHEVSKY; ROTHERMEL, 2002; MALISHEVSKY; ROTHERMEL; ELBAUM, 2002; ROTHERMEL et al., 1999; ROTHERMEL et al., 2001; ROTHERMEL et al., 2002).

Vários outros trabalhos são baseados nessa informação. Li et al. (2010) analisam cinco algoritmos para otimização do teste de regressão, sendo três baseados em cobertura de requisitos, um no algoritmo *Hill Climbing*; e um na aplicação de algoritmos genéticos. Os resultados foram avaliados com base no percentual médio de cobertura de Requisitos.

A simulação indicou que dois algoritmos baseados em cobertura de requisitos levaram vantagem. Esse trabalho apresenta similaridades com (LI; HARMAN; HIERONS, 2007), que utiliza as mesmas técnicas, mas avalia o resultado considerando as métricas APBC (*Average Percentage Block Coverage*), APDC (*Average Percentage Decision Coverage*) e APSC (*Average Percentage Statement Coverage*).

Jeffrey e Gupta (2008) apresentam uma abordagem para a priorização de casos de teste baseada em requisitos de cobertura presentes nas fatias relevantes (trechos que influenciam ou podem influenciar a saída de um programa quando executam em um caso de teste específico) das saídas dos casos de teste. As modificações que afetam a saída de um programa em determinado caso de teste também devem afetar alguma computação na fatia relevante para aquele caso de teste. Com base nessa afirmação, a abordagem prioriza os casos de teste atribuindo peso maior aos casos de teste com maior número de *branches* em sua fatia relevante da saída. Os resultados foram avaliados pela taxa de detecção de falhas.

Um critério bastante empregado em trabalhos de otimização do teste de regressão é o histórico de detecção de falhas dos testes. O teste de regressão envolve a reutilização de testes realizados em estágios anteriores do desenvolvimento, e é viável registrar casos de teste que detectaram falhas anteriormente. Apesar da ausência de garantias de que casos de teste que revelaram falhas de fato possuam maior potencial para a detecção de falhas, esses casos de teste podem ser considerados como tendo poder comprovado de detecção (HARMAN, 2011). Outros trabalhos empregam o histórico de falhas na otimização do teste de regressão, dentre os quais podemos destacar (KIM; PORTER, 2002) e (KIM; BAIK, 2010).

Black, Melachrinoudis e Kaeli (2004) propõem um modelo de Programação Linear Inteira para a minimização de casos de teste contendo dois objetivos: otimizar o nível de cobertura do código maximizando a taxa de detecção de falhas. O trabalho emprega conjuntos de associações definição-uso (*definition-use association* - DUA) para definir um conjunto de elementos requeridos para o processo de teste. A avaliação do modelo mostrou que a utilização da informação de detecção pregressa de falhas trás benefícios ao desempenho da minimização, e que as *suites* minimizadas com relação a uma coleção de falhas do programa são eficazes em revelar falhas subsequentes do programa.

Gao, Guo e Zhao (2015) propõe um algoritmo de otimização por colônia de formigas para priorização de casos de teste baseado em três fatores: número de falhas detectadas, tempo de execução e severidade das falhas. O objetivo foi maximizar a taxa de detecção de falhas.

Yoo e Harman (2007) empregaram o histórico de detecção de falhas em uma abordagem multiobjetivo que além das falhas considerou a cobertura do código e o tempo de execução dos testes. A pesquisa realiza estudos empíricos utilizando duas formulações:

uma com dois objetivos (maximizar cobertura e minimizar custo nos subconjuntos de casos de teste) e outra com três objetivos (os dois anteriores, e maximizar a cobertura de falhas passadas). O trabalho compara as soluções geradas para as duas formulações por duas abordagens variantes do NSGA-II (*Non-dominated Sorting Generic Algorithm - II*) e por algoritmos gulosos, e os resultados indicam que para programas pequenos as abordagens multiobjetivo superaram os algoritmos gulosos.

Além da seleção e da minimização, alguns trabalhos sobre priorização de casos de teste também se beneficiaram do emprego de informações do histórico de falhas. Qu et al. (2007) apresenta e avalia uma técnica para teste caixa-preta de regressão. Foi empregada uma matriz que representa a relação existente entre os casos de teste, em termos de detecção de falhas. A técnica proposta agrupou os casos de teste com base nos tipos de falhas que eles revelam, e era capaz de ajustar dinamicamente suas prioridades, escolhendo um caso de teste e escalando ou de-escalando a prioridade dos casos de teste ligados a ele. Os resultados foram avaliados com base na taxa de detecção de falhas e evidenciaram a eficácia da técnica em ambientes de caixa-preta.

Ainda seguindo a conjectura de que casos de teste com propriedades em comum possuem capacidades similares de detecção de falhas, Carlson, Do e Denton (2011) implementa técnicas de priorização que incorporam uma abordagem de clusterização e emprega, além da cobertura do código e dados reais do histórico de falhas, a complexidade do código calculada com base em duas informações: Quantidade de linhas de código e contagem de dependências dos métodos da classe. Os resultados apontam que a priorização que utiliza uma abordagem de clusterização pode melhorar a eficácia das técnicas de priorização de testes, além de ser eficaz sob restrições de tempo.

Além de informações relacionadas ao código e ao histórico de execução dos testes, já houveram iniciativas no sentido de priorizar os casos de teste de acordo com sua importância para o negócio. Khandai, Acharya e Mohapatra (2011) empregam a métrica BCV (*Business Criticality Value*), que representa a criticidade (contribuição para o negócio) das funções do programa. A abordagem proposta no trabalho é composta de três etapas. A primeira consiste na manutenção de um repositório de projetos. Na segunda etapa, cada novo projeto é comparado aos existentes no repositório, e à partir de dados estatísticos as funções alteradas no novo projeto recebem valores de criticidade. Os casos de teste são, por fim, ordenados de acordo com sua criticidade no negócio. Acharya, Khandai e Mohapatra (2012) também empregam a métrica BCV na priorização de casos de teste. Os dois trabalhos não necessitam de informações sobre o histórico de detecção de falhas, o que se mostra útil principalmente no início do desenvolvimento do software. Porém, ambos os trabalhos presumem a existência de um repositório de projetos para a obtenção da criticidade das funções.

Apesar de um grande número de estudos aplicar algoritmos gulosos ao problema do

teste de regressão, essa é uma área também propícia à aplicação de técnicas de otimização. Singh, Kaur e Suri (2010) avaliam a aplicação da otimização por colônia de formigas no problema da priorização de casos de teste. Os resultados da priorização obtidos com o algoritmo foram comparados com outras ordenações das *suites*, dentre as quais podemos mencionar a ordem original, uma ordem aleatória, a ordem reversa e a ordem ótima. A avaliação foi realizada com uma *suite* fictícia, empregando a métrica APFD. Os resultados indicaram que a ordenação obtida com a otimização por colônia de formigas obteve resultados próximos da ordenação ótima.

Suri e Singhal (2015) implementam uma técnica de priorização e seleção de casos de teste baseada em ACO, e analisa o uso de restrições de tempo. Observou-se que, quanto mais alta a restrição de tempo, maiores as chances de obtenção de uma *suite* de testes ótima (com um maior número de melhores caminhos e menor tempo de execução).

2.1 Considerações sobre os trabalhos

Gill e Sikka (2011) aponta a relação existente entre oito métricas orientadas ao design e a susceptibilidade a falhas de um componente do software. Apesar da relevância dessas métricas, não foram encontrados registros da aplicação dessas métricas em estudos relacionados à otimização do teste de regressão, apesar de serem de obtenção relativamente simples por meio de instrumentação do código.

Conforme dito anteriormente existe um grande número de trabalhos que emprega a cobertura de código, especialmente como objetivo intermediário. Entretanto, Hao et al. (2016) alerta para o fato de que as técnicas podem apresentar resultados melhores caso não estejam baseadas exclusivamente nesse objetivo. Yoo e Harman (2007) destacam os benefícios em empregar mais de um critério de teste nas técnicas de otimização de teste de regressão. Uma vez que o resultado do teste de regressão só é conhecido após sua realização e não existe uma ligação precisa entre cobertura de código e detecção de falhas, é interessante complementar a cobertura dos casos de teste com outros critérios e objetivos (HARMAN, 2011). Diversas outras informações são usadas na otimização do teste de regressão, como o histórico de detecção de falhas, custo de execução dos casos de teste, criticidade para o negócio, susceptibilidade a falhas, etc. (CATAL; MISHRA, 2013).

O histórico de falhas é bastante empregado em técnicas de otimização do teste de regressão. Alguns trabalhos que empregam essa técnica utilizam essa informação como métrica para avaliação da qualidade das soluções obtidas. No entanto, apenas uma fração dos trabalhos empregam essa informação para guiar o processo de busca. Existe uma métrica que indica a taxa de detecção de falhas, o APFD, que é a mais empregada em trabalhos relacionados ao teste de regressão, e será utilizada na avaliação das soluções da abordagem proposta.

Uma parcela representativa dos trabalhos sobre otimização de teste de regressão emprega técnicas gulosas, visando antecipar a detecção das falhas construindo soluções baseadas na sucessiva escolha de testes que satisfaçam ótimos locais para determinados critérios. Porém, já há diversos trabalhos usufruindo da utilização de técnicas de busca, como algoritmos genéticos e otimização por colônia de formigas. O emprego de metaheurísticas pode promover uma melhor exploração do espaço de soluções e a obtenção de melhores resultados.

3 Uma Abordagem Híbrida para Priorização e Seleção de Casos de Teste

Neste capítulo é apresentada a abordagem proposta por Silva et al. (2016) para a realização do teste de regressão. A abordagem é baseada em um híbrido de técnicas de seleção e priorização de casos de teste e busca, a partir do conjunto inicial de casos de teste de um programa, obter um subconjunto cujos elementos respeitem determinada restrição de tempo obtendo o máximo ganho possível, expresso na forma da criticidade total dos casos de teste selecionados. Apesar de essa abordagem não ser o objeto principal deste trabalho, constitui um ponto de partida para a elaboração da abordagem desta dissertação, descrita no Capítulo 4.

A abordagem modela a seleção de casos de teste na forma do problema da mochila binária (*0-1 knapsack problem*) (KELLERER; PFERSCHY; PISINGER, 2004), onde dado um conjunto de itens que possuem peso (custo) e valor (ganho), deve-se encher uma mochila de forma a alcançar um valor máximo total, respeitando um limite máximo de peso. No contexto da proposta, os itens a serem dispostos na mochila são casos de teste. O peso do item é expresso por seu tempo de execução, e o valor do item é expresso por sua criticidade.

O trabalho considera a criticidade de um caso de teste como sendo um indicativo de o quão importante é a sua execução dentro do contexto do sistema. A ideia por trás desse mecanismo é a de que importância de execução de um caso de teste é relacionada à importância e potencial de falha dos componentes de software que ele cobre. Essa métrica é utilizada posteriormente para guiar o processo de seleção, permitindo a obtenção de uma *suite* de testes que melhor represente as prioridades da equipe de desenvolvimento.

Apesar de existirem trabalhos que empregam o *Business Criticality Value* (BCV) das funções do programa como chave para a priorização (ACHARYA; KHANDAI; MOHAPATRA, 2012; KHANDAI; ACHARYA; MOHAPATRA, 2011), não foram encontrados relatos na literatura sobre como atribuir um valor de criticidade ou o grau de importância de se executar um caso de teste. Para obter de forma sistemática a criticidade dos casos de teste e resolver o problema do teste de regressão foi então proposta uma abordagem composta de cinco etapas, detalhadas nas próximas seções (Figura 5):

- Etapa 1 - cálculo da relevância das classes;
- Etapa 2 - inferência da criticidade das classes;
- Etapa 3 - cálculo da criticidade dos casos de teste;

- Etapa 4 - seleção dos testes;
- Etapa 5 - priorização dos testes.

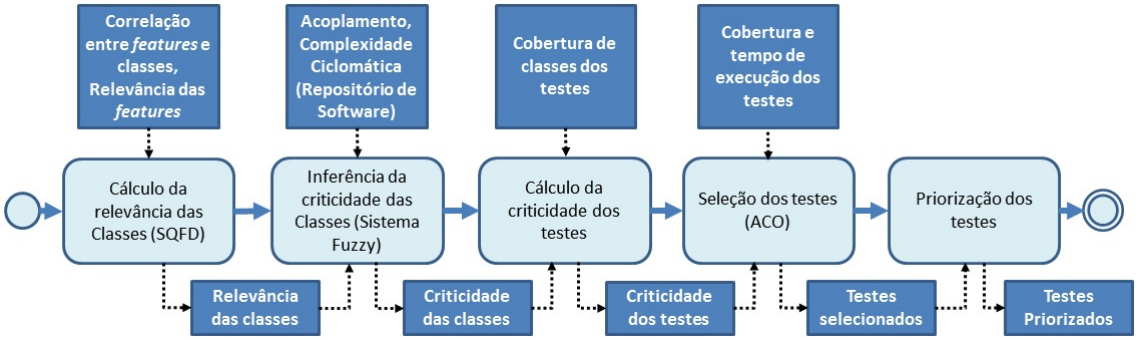

Figura 5 – Abordagem proposta em (SILVA et al., 2016)

3.1 Etapa 1 - Cálculo da relevância das classes

Conforme mencionado anteriormente, a criticidade de execução de um caso de teste é um conceito de caráter interpretativo, variável e bastante sujeito ao contexto do desenvolvimento, o que inviabiliza sua definição de maneira exata. A abordagem considera, então, que a execução de um caso de teste está ligada à importância de se verificar cada componente do software coberto pelo caso de teste em questão.

Uma das entradas empregadas pela abordagem no cálculo da criticidade dos componentes é a relevância dos componentes, cujo valor é obtido por meio do SQFD (*Software Quality Function Deployment*) (HAAG; RAJA; SCHKADE, 1996), um processo estruturado de desenvolvimento de produtos de software. SQFD é focado nas necessidades do cliente, assegurando que as *features* do produto não são determinadas pelo que é ou não tecnicamente possível, mas pelas demandas do usuário. SQFD foi utilizado no mapeamento da importância das *features* nas classes do software (aspectos técnicos de um produto de software). Essa etapa possui duas entradas:

- Relevância das *features*: nível de interesse que o cliente atribui às funcionalidades do sistema, em uma escala de 0 a 10;
- Correlação entre as *features* e as classes: pode ser igual a 0 (ausência de correlação), 1 (possível correlação), 3 (correlação fraca) e 9 (correlação forte), de acordo com as prescrições do SQFD (HAAG; RAJA; SCHKADE, 1996).

O cálculo da relevância ocorre em três passos. No primeiro, as *features* do software e suas importâncias (representadas por valores de 0 a 10) são listadas. O segundo passo requer a especificação do nível de correlação entre as *features* e classes do software por meio

de uma ferramenta desenvolvida pelos próprios autores que semi-automatizou o processo. A equipe de desenvolvimento teve apenas que apontar a classe principal de cada *feature*. Depois disso, a ferramenta atribui correlação 9 à classe principal de cada *feature*, correlação 3 a cada classe referenciada pelas classes principais, correlação 1 às classes referenciadas pelas classes com correlação 3, e recursivamente continua essa tarefa, atribuindo correlação 1 às outras classes, até a exaustão da busca.

O passo final dessa etapa é o cálculo da relevância de cada classe do software. A Equação 3.1 apresenta a fórmula utilizada na obtenção da relevância de uma classe.

$$R_j = \sum_{i=1}^{n} V_{ij} \cdot F_i \tag{3.1}$$

Onde:

- R_j é a relevância da classe j;
- i é o índice de uma *feature*
- n é o número de *features*;
- V_{ij} é a correlação entre a *feature* i e a classe j; e
- F_i é a relevância da feature i.

3.2 Etapa 2 - Inferência da criticidade das classes

Devido à anteriormente citada inexistência de uma definição formal para a obtenção da criticidade dos casos de teste, a abordagem obtém esse valor com base na criticidade dos componentes de software que esses casos de teste cobrem. Quanto mais crítico o componente é, mais importante é a execução dos testes que o contemplam.

Para obter essa criticidade dos componentes, foi utilizado um sistema de inferência *fuzzy* de Mamdani (MAMDANI, 1977). Os três valores *crisp* empregados na entrada do sistema de inferência foram:

- Acoplamento: grau de interdependência de uma classe com cada uma das outras classes;
- Complexidade Ciclomática: quantidade de caminhos de execução possíveis dentro de um trecho de código, que pode abranger desde uma simples função ou método até a extensão total do programa (MCCABE, 1976);
- Relevância da classe: obtida da Etapa 1.

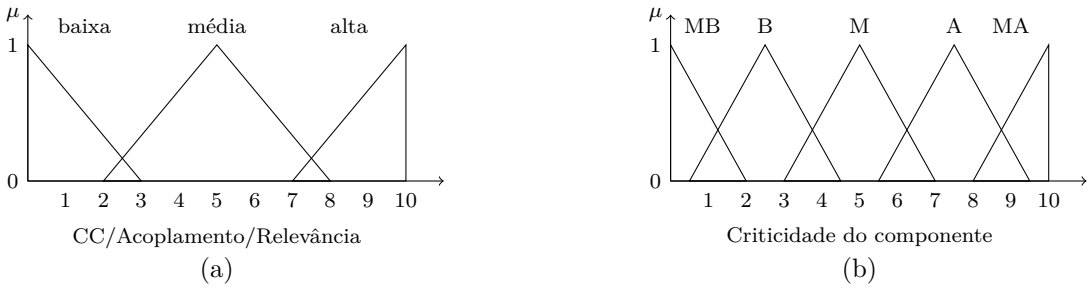

Figura 6 – Distribuição dos conjuntos *fuzzy* para as variáveis de entrada (a) e de saída (b)

Tabela 1 – Base de regras empregada na abordagem

	Relevância / Complexidade Ciclomática								
Acoplamento	A/A	A/M	A/B	M/A	M/M	M/B	B/A	B/M	B/B
A	MA	MA	M	A	M	M	M	M	B
M	MA	A	M	M	M	M	M	B	B
B	A	M	M	M	M	B	B	B	MB

É interessante observar que o sistema contempla tanto métricas do software ligadas à susceptibilidade a falhas (GILL; SIKKA, 2011) quanto uma mensuração da importância que é atribuída ao objeto dentro do sistema. A relevância não reflete a susceptibilidade a falhas de um componente do software, mas em compensação permite que seja priorizado o que é mais importante para o usuário final. A variável de saída do sistema de inferência *fuzzy* é a criticidade do componente (CC). Quanto mais alta a criticidade, maior a relevância dos testes que a cobrem. Tanto os três valores utilizados na entrada quanto a CC na saída são normalizados no intervalo de 0 a 10.

Foram utilizadas funções de pertinência triangulares para mapear os conjuntos *fuzzy* nas variáveis de entrada. Para cada variável de entrada, foram mapeados três conjuntos *fuzzy*: Baixo (B), Médio (M) e Alto (A). Também foram empregadas funções de pertinência triangulares para mapear os seguintes conjuntos *fuzzy* à variável de saída: Muito Baixo (MB), Baixo (B), Médio (M), Alto (A) e Muito Alto (MA) (Figura 6).

Para gerar a criticidade das classes, foi implementada uma base de 27 regras para o sistema de inferência *fuzzy* proposto, que foi refinada com a opinião de especialistas até a aprovação de seus resultados (Tabela 1).

3.3 Etapa 3 - Cálculo da criticidade dos casos de teste

Catal e Mishra (2013) já haviam evidenciado as vantagens de se considerar a cobertura dos casos de teste como forma de agilizar a detecção de falhas. Na abordagem proposta, foram considerados mais críticos os casos de teste que cobrem componentes de software tidos como mais críticos. Nessa etapa, a criticidade de cada caso de teste é

calculada com base em sua respectiva cobertura de classes, obtida por meio da análise das informações dos testes *jUnit* com a ferramenta *EclEmma*[1], que realiza análise do código diretamente na IDE Eclipse[2]. Para obter a criticidade de um caso de teste k (TC_k), é aplicada a seguinte fórmula:

$$TC_k = \frac{\sum_{j=1}^{m}(CC_j) * x_{jk}}{\sum_{j=1}^{m} CC_j} \tag{3.2}$$

Onde:

- TC_k é a criticidade do caso de teste k;
- j é o índice da classe;
- x_{jk} é a cobertura da classe j pelo caso de teste k;
- m é o número de classes cobertas pelo teste considerado; e
- CC_j é a criticidade da classe j.

3.4 Etapa 4 - Seleção dos testes

Nessa etapa, são aplicadas as restrições de tempo à *suite* de testes. As entradas são o conjunto de casos de teste do sistema, e a saída é um conjunto de testes, cuja cardinalidade depende de uma restrição de tempo. Isso caracteriza a técnica de seleção de casos de teste.

A etapa de seleção se divide em dois passos. O primeiro é a descoberta de todos os casos de teste que podem ser executados, por meio de uma análise do histórico do repositório de código abrangendo o período a partir da última alteração do software. Mesclando essa lista à informação de cobertura dos casos de teste, é possível descobrir todos os casos ligados a pelo menos um arquivo modificado na última versão do software. Esses casos são candidatos a execução.

Porém, o tempo requerido para a execução desses testes pode ser maior que o disponível. É importante enfatizar que a abordagem funciona a nível de classe de teste. Uma classe de teste pode possuir vários casos de teste que testam regras lógicas relacionadas a um componente específico de software. O segundo passo dessa etapa é a resolução do problema da mochila binária mencionado anteriormente. Foi utilizada otimização por colônia de formigas, e o algoritmo implementado foi o MMAS (*Max-Min Ant System*) (STÜTZLE; HOOS, 2000), um algoritmo de otimização por colônia de formigas que explora

[1] http://www.eclemma.org/
[2] http://www.eclipse.org/

o espaço de soluções evitando estagnação precoce. A formulação matemática empregada foi:

$$Maximize : \sum_{k=1}^{t} TC_k * x_k \tag{3.3}$$

$$Restrição : \sum_{k=1}^{t} ET_k * x_k \leq KLC \tag{3.4}$$

Onde:

- k é o índice de um caso de teste;
- t é o número de casos de teste participantes da etapa de seleção;
- TC_k é a criticidade (ganho) do caso de teste k;
- $x_k = 0$ quando o caso de teste k não foi colocado na mochila;
- $x_k = 1$ quando o caso de teste k foi colocado na mochila;
- ET_k é o tempo de execução (peso) do caso de teste k; e
- KLC é a capacidade de carga total da mochila (restrição de tempo).

Os passos do algoritmo são resumidos a seguir:

1. **Inicialização dos parâmetros e variáveis**: são especificados o número de formigas (n_a), a concentração inicial de feromônio (τ_0), a restrição de tempo (KLC), α (importância da informação feromonal), β (importância da informação heurística), dentre outros parâmetros. Além disso, nesse passo cada formiga é posicionada em um caso de teste, a partir do qual ela irá iniciar o seu caminho (construção da solução).
2. **Construção da solução**: as soluções são construídas pelas formigas por meio da seleção iterativa de testes, considerando uma probabilidade de transição que é maior para testes que aparentemente levem a melhores resultados. A probabilidade de seleção do próximo teste l por uma formiga k é obtida por meio da Equação 3.5

$$P_k^l = \frac{[\tau_l]^\alpha * [\eta_l]^\beta}{\sum_{A \in allowed(k)} [\tau_A]^\alpha * [\eta_A]^\beta} \tag{3.5}$$

onde:

- α representa a importância da informação feromonal;
- β representa a importância da informação heurística;

- τ_l representa a quantidade de feromônio associada ao caso de teste l;
- A representa os casos de teste por onde a formiga k ainda não passou, mas que se ajustam à restrição de tempo do problema; e
- η_l representa a informação heurística do teste l, obtida por meio da Equação 3.6,

$$\eta_l = \frac{TC_l}{ET_l} \tag{3.6}$$

onde TC_l e ET_l equivalem à criticidade e o tempo de execução do caso de teste l, respectivamente. Um valor alto de criticidade, combinado a um baixo tempo de execução, oferecem um bom indicativo de que o caso de teste é uma boa escolha.

3. **Atualização do feromônio**: a atualização do feromônio em um momento t para um caso de teste equivale à quantidade de feromônio incrementada após o depósito, de acordo com a seguinte equação:

$$\tau_{tc_l}(t+1) = \rho\tau_{tc_l}(t) + \Delta\tau_{tc_l}(t) \tag{3.7}$$

onde ρ representa a persistência do feromônio. Vale observar que a concentração inicial do feromônio é representada pela constante $\tau_{(0)}$. O depósito total de feromônio em um caso de teste considera o depósito de cada formiga que o selecionou, conforme mostra a Equação 3.8.

$$\Delta\tau_{tc_l}(t) = \sum_{k=1}^{n\_a} \Delta\tau^k_{tc_l}(t) \tag{3.8}$$

Por fim, o depósito de feromônio feito por uma formiga é representado por:

$$\Delta\tau^k_{tc_l}(t) = t_l * \frac{Q}{T_k} \tag{3.9}$$

onde t_l representa o tempo de execução do caso de teste l, Q é uma constante positiva e T_k é o somatório dos tempos de execução de todos os S_k testes selecionados, ou seja:

$$T_k = \sum_{i=1}^{S_k} T_i \tag{3.10}$$

4. **Avaliação da solução**: após a construção por uma formiga, a qualidade da solução é avaliada por sua criticidade total, representada por:

$$C_k = \sum_{i=1}^{S} TC_i \tag{3.11}$$

Onde C_k é a criticidade total da solução, S é o número de casos de teste da solução, e TC_i é a criticidade do i-ésimo caso de teste da solução

3.5 Etapa 5 - Priorização dos testes

A última etapa da abordagem é a priorização dos testes. Na etapa 4 já foram selecionados os testes que deveriam ser executados respeitando a restrição de tempo, e o resultado da melhor formiga foi armazenado. A priorização dos testes consiste em dispor a seleção de testes dessa formiga, em ordem decrescente, utilizando a criticidade como chave.

3.6 Resultados e Limitações

A avaliação empírica da abordagem foi realizada por meio da comparação dos resultados alcançados por ela com os obtidos em duas variações da abordagem onde variou o algoritmo de seleção na Etapa 4. Em uma variação, o algoritmo ACO foi substituído por uma busca exaustiva, e na outra, por um algoritmo guloso.

A busca exaustiva (método força bruta) consiste em enumerar e verificar sistematicamente todas as soluções candidatas possíveis para o problema. Essa abordagem garante que a melhor solução possível para o problema será encontrada. Porém, apresenta a desvantagem de, conforme o espaço de busca torna-se mais amplo, requerer cada vez mais tempo para a execução. A utilização dessa técnica permitiu melhor avaliar os resultados obtidos com a abordagem proposta.

Os algoritmos gulosos empregam a heurística de buscar os ótimos locais (a melhor solução para o momento) a cada etapa, visando atingir um ótimo global. No experimento, a heurística utilizada foi adicionar sequencialmente o teste com o maior valor de criticidade que ainda estivesse disponível enquanto houvessem testes que pudessem ser inseridos na solução sem violar a restrição de tempo. Apesar de ser comum a estratégia gulosa não obter necessariamente uma solução ótima, seus resultados costumam ser satisfatórios, e ela funciona em um tempo razoável. isso tornou-a uma boa alternativa para testar a qualidade da solução proposta.

O critério empregado na comparação entre as três abordagens foi então a criticidade total da *suite* de testes obtida como saída da abordagem. Todos os dados dos requisitos e classes, assim como a correlação existente entre eles, foram fictícios e gerados de forma sistemática, assim como os cenários de aplicação da abordagem.

A abordagem foi executada 30 vezes para cada configuração (quantidade de testes/algoritmo/limite de tempo), e a Tabela 2 apresenta a média dos resultados obtidos para cada configuração. É possível observar que o tempo de execução da busca exaustiva apresentou um comportamento exponencial. Ao tempo em que é possível perceber que os resultados da abordagem (ACO), em termos de criticidade, se aproximaram da busca exaustiva, mas obtiveram menor tempo de execução à medida em que o problema se tornava maior. Os algoritmos gulosos apresentaram o melhor tempo de execução, porém

Tabela 2 – Dados dos Experimentos

Testes	Limite de tempo	Algoritmo	Criticidade	Tempo est. exec. testes (ms)	Tempo est. exec. Alg. (ms)
5	6263	Guloso	100,7389	5882	50
		ACO	100,7389	5882	200
		Exaustivo	100,7389	5882	179
10	15234	Guloso	98,24	11822	41
		ACO	151,5102	15016	31694
		Exaustivo	151,5102	15016	2248
15	17863	Guloso	201,5038	16093	72
		ACO	309,2301	16152	131214
		Exaustivo	309,2301	16152	44290
20	21627	Guloso	248,1591	21470	69
		ACO	407,8005	21376	80716
		Exaustivo	407,8005	21376	683745
25	35755	Guloso	348,5454	35627	74
		ACO	405,8741	35011	103460
		Exaustivo	406,6174	35702	24127799
30	37096	Guloso	401,1047	36251	113
		ACO	611,1333	35610	238933
		Exaustivo	611,7414	36709	906554914
35	40262	Guloso	552,6803	40179	142
		ACO	714,6189	39814	374406
		Exaustivo	-	-	-
40	49169	Guloso	552,0443	47523	155
		ACO	715,0143	48342	448349
		Exaustivo	-	-	-

resultados bastante inferiores aos dos outros algoritmos. Além disso, conforme esperado, a busca exaustiva foi capaz de obter os resultados ótimos para o problema, mas não pôde ser concluída quando o espaço de busca envolvia a existência de mais de 30 testes. Já a abordagem proposta, apesar de não conseguir encontrar a solução ótima quando o espaço de busca passou a envolver 25 testes foi capaz de continuar obtendo bons resultados, conforme o espaço de busca das soluções crescia.

3.7 Contribuições e Limitações

A abordagem proposta trouxe várias contribuições, como:

- A utilização da importância dos componentes do software para o usuário como variável na obtenção do conjunto de testes de regressão, por meio do mapeamento da correlação entre a importância das *features* para os clientes e a relevância das classes do sistema, empregando SQFD;

- Um mecanismo de inferência da criticidade das classes do software empregando métricas conhecidas relacionadas ao desenvolvimento de software;

- Um método para obter a criticidade das classes de teste do programa por meio da criticidade das classes por ele cobertas;

- Um mecanismo de seleção de testes utilizando otimização por colônia de formigas;

- Um mecanismo de priorização de casos de teste que emprega a criticidade como chave;
- Uma avaliação da abordagem utilizando dados simulados.

Entretanto, é necessário listar algumas limitações do trabalho, como:

- A etapa de priorização de casos de teste, realizada no final da abordagem, consiste apenas de uma ordenação dos casos de teste que tomava por chave a criticidade. Apesar de um grande número de técnicas de priorização ser baseada em algoritmos gulosos, o processo de ordenação dos testes poderia se beneficiar da aplicação de uma metaheurística, que já fazia parte da etapa de seleção;
- A ordenação final da *suite* de testes era realizada somente em função da criticidade dos casos de teste. Apesar de válida, a abordagem poderia ser melhorada se a avaliação dos resultados produzidos considerasse também outras características dos testes que compõem o resultado, como a potencial taxa de detecção de falhas da *suite* gerada;
- Outra limitação da abordagem decorre da própria utilização da técnica de seleção, que ao suprimir a execução de casos de teste, pode impedir a detecção de falhas de determinado tipo;
- Mostrou-se necessária uma melhor experimentação também no sentido de avaliar a aplicabilidade da proposta em situações mais verossímeis de teste de regressão. Os experimentos para validação da abordagem envolviam somente um programa exemplo didático, gerados sistematicamente pelos pesquisadores e não são suficientemente representativos da realidade. A experimentação poderia ser enriquecida se houvesse o emprego de *datasets* públicos, empregados em outros trabalhos, ou mesmo de programas reais.

Para sanar as limitações aqui apresentadas, este trabalho propõe uma atualização da abordagem. As modificações realizadas estão descritas no Capítulo 4, e a experimentação realizada no Capítulo 5.

4 Abordagem Proposta

A abordagem proposta por Silva et al. (2016) e exposta no Capítulo 3 emprega um híbrido de seleção e priorização de casos de teste para lidar com o teste de regressão. Conforme mencionado, apesar de suas contribuições essa abordagem deixa em aberto alguns espaços para melhorias. Este capítulo apresenta algumas modificações realizadas na abordagem, que passa a ser composta de três etapas, e baseada na priorização de casos de teste (Figura 7) .

Não existe uma maneira padronizada de definir um caso de teste como sendo de execução mais crítica do que outro. Visto que o teste de software tem por principal objetivo encontrar falhas no sistema, poder-se-ia definir como crítico um caso de teste que detecta falhas. Infelizmente, na prática a informação de quais casos irão encontrar falhas pode não estar disponível com antecipação, mas as abordagens para o teste de regressão costumam se basear em alguns critérios.

Um dos principais critérios é a cobertura do código (CATAL; MISHRA, 2013). As técnicas que se baseiam na maximização da cobertura presumem que, dada a necessidade de executar um trecho do código para encontrar as falhas nele contidas, uma sequência que vasculhe a maior quantidade de código no menor tempo possível apresenta maior probabilidade de varrer rapidamente trechos do programa com falhas. Porém, a simples cobertura do código não é garantia de melhoria na detecção de falhas (HAO et al., 2016), e é recomendado empregá-la na otimização do teste de regressão em conjunto com outras métricas (YOO; HARMAN, 2007).

A abordagem proposta (Figura 7) considera para a priorização dos casos de teste não somente a cobertura dos casos de teste, mas também a criticidade de se verificar esses componentes. A abordagem é então composta de três etapas, que iniciam com a inferência da criticidade dos componentes de software, que é utilizada para obter a criticidade dos casos de teste, que por sua vez é empregada na priorização dos casos de teste.

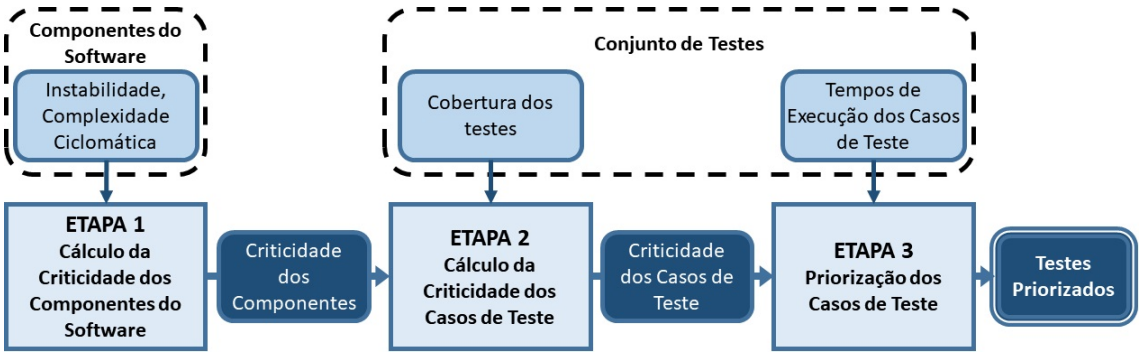

Figura 7 – Abordagem Proposta

4.0.1 Etapa I - Cálculo da criticidade dos componentes de Software

Nessa etapa, a criticidade de cada componente de software é inferida. Uma vez que não existem funções ou métodos específicos para obter essa métrica, foi utilizado um sistema de inferência *fuzzy* para mapear o conhecimento dos especialistas relacionado à importância de se executar testes em determinados componentes de software. As entradas desse sistema são métricas extraídas dos componentes do software, diretamente ligadas à susceptibilidade a falhas de um componente (GILL; SIKKA, 2011). As métricas empregadas foram:

- Complexidade Ciclomática (CC) (MCCABE, 1976): essa métrica indica a complexidade do código na forma do número de possíveis caminhos de execução;
- Instabilidade (MARTIN, 2003): nível de relacionamento entre os componentes do software. O valor da instabilidade é determinado pela expressão a seguir:

$$I = \frac{C_e}{C_e + C_a} \tag{4.1}$$

onde C_e é o acoplamento eferente (número de dependências do componente, ou seja, o número de componentes a quem o componente avaliado faz referência); e C_a é o acoplamento aferente (número de componentes que dependem do componente, ou seja, o número de componentes que referenciam o componente avaliado).

A variável de saída do sistema de inferência *fuzzy* proposto é a criticidade do componente de código. A base de regras (Tabela 3), as entradas e as saídas do sistema de inferência *fuzzy* foram definidas considerando o nosso conhecimento até então, e validada com especialistas da área de engenharia de software (Figura 8).

Tabela 3 – Base de regras do sistema de inferência *fuzzy*

		Instabilidade				
		Muito Baixa	**Baixa**	**Média**	**Alta**	**Muito Alta**
CC	**Muito Baixa**	Muito Baixa	Muito Baixa	Baixa	Baixa	Média
	Baixa	Muito Baixa	Baixa	Baixa	Média	Média
	Média	Baixa	Baixa	Média	Alta	Alta
	Alta	Baixa	Média	Alta	Alta	Muito Alta
	Muito Alta	Média	Média	Alta	Muito Alta	Muito Alta

4.0.2 Etapa II - Cálculo da criticidade dos casos de teste

Assim como em (SILVA et al., 2016), a abordagem proposta define que a execução de um caso de teste é tão crítica quanto os componentes de software por ele cobertos o são. Desta forma, a criticidade TC de cada caso de teste j é calculada utilizando como

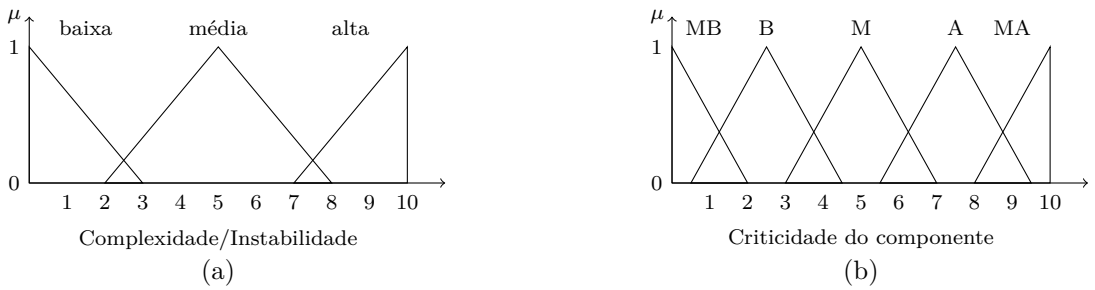

Figura 8 – Distribuição dos conjuntos *fuzzy* para as variáveis de entrada (a) e de saída (b)

entrada a criticidade dos componentes cobertos por ele e sua informação de cobertura, conforme representado pela fórmula:

$$TC_j = \frac{\sum_{i=1}^{nj} FC_i * x_{i,j}}{\sum_{i=1}^{nj} FC_i} \tag{4.2}$$

onde:

- i: índice de um componente coberto pelo caso de teste j;
- nj: número de componentes a serem testados;
- FC_i: criticidade do componente i;
- $x_{i,j}$: cobertura do componente i pelo caso de teste j (valor de 0 a 1).

4.0.3 Etapa III - Priorização dos casos de teste

Nessa etapa, os casos de teste disponíveis são priorizados utilizando otimização por colônia de formigas. As principais entradas empregadas nessa etapa são a criticidade (obtida na Etapa II), o histórico de detecção de falhas e o tempo de execução dos casos de teste. Ou seja, esta etapa sugere uma ordem de execução que leve em conta a criticidade (com o intuito de escolher os casos de teste com maior potencial de detecção de falhas), assim como a capacidade de encontrar falhas (atestado pelo seu histórico de detecção) e o tempo de execução do caso de teste (com o objetivo de dar preferência aos que encontram falhas demandando uma menor quantidade de tempo).

O processo de priorização foi modelado como uma adaptação do problema do caixeiro viajante (REINELT, 1994), que consiste em encontrar o menor caminho fechado por um conjunto de cidades, visitando cada cidade exatamente uma vez. Nesse caso, as cidades seriam representadas pelos casos de teste, e cada caso seria conectado com todos os outros (grafo completo).

Para resolver esse problema, assim como na abordagem original foi empregado o *MAX-MIN Ant System* (MMAS), algoritmo de otimização por colônia de formigas que já

foi descrito no Capítulo 3. A principal diferença está no passo 4 do algoritmo (descrito na Seção 3.4), que avalia as soluções com base em duas métricas descritas a seguir: o APFD e o $APFD_C$. Os parâmetros de funcionamento do algoritmo e o processo de escolha da equação da informação heurística dos casos de teste estão descritos no Capítulo 5.

Avaliação da qualidade das soluções

Conforme dito anteriormente, além de a abordagem visar a priorização de casos de teste considerados críticos, ela também deverá considerar a taxa de detecção de falhas do conjunto de testes. Para quantificar esta eficiência, a literatura apresenta duas métricas bastante empregadas em trabalhos de priorização de casos de teste: o APFD e o $APFD_C$.

O APFD (*Average Percentage of Faults Detected*) (ELBAUM; MALISHEVSKY; ROTHERMEL, 2002; ELBAUM; MALISHEVSKY; ROTHERMEL, 2000) é uma métrica que representa a taxa de em que as falhas são detectadas por uma *suite* de testes. Mais especificamente, o APFD verifica que caso de teste (na ordem em que aparece no conjunto de testes) encontra primeiro cada falha. Quanto maior o seu valor para um conjunto de testes, mais eficiente é a detecção de falhas desse conjunto. Na obtenção do valor de APFD é utilizado o histórico de detecção de falhas dos testes. Mais especificamente, é verificada a posição no conjunto total de testes do caso de teste que encontrou primeiro cada falha.

A fórmula do cálculo do APFD é definida por:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_i + ... + TF_m}{nm} + \frac{1}{2n} \tag{4.3}$$

Onde:

- m – número de falhas;
- n – número de casos de teste; e
- TF_i – posição do primeiro caso de teste que detectou a falha i

A Tabela 4 apresenta um exemplo de matriz de falha fictícia, na qual as intersecções coluna X linha indicam a detecção ou não de uma falha por determinado caso de teste. Este modelo será utilizado para ilustrar o cálculo da obtenção do APFD.

No exemplo, uma *suite* composta pelos testes 3, 5, 9, 4, 6 e 1 teria o seu valor de APFD definido por:

$$APFD = 1 - \frac{1+4+6+2+2+3+1+4+2+4}{10*10} + \frac{1}{2*10} = 0,66 \tag{4.4}$$

Tabela 4 – Exemplo de Matriz de Falhas

	Falhas										
	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_10	Total
Teste 1			X		X	X					3
Teste 2				X		X		X	X		4
Teste 3	X						X				2
Teste 4		X				X		X		X	4
Teste 5				X	X				X		3
Teste 6								X			1
Teste 7		X									1
Teste 8			X	X			X	X		X	5
Teste 9	X			X		X			X		4
Teste 10		X			X					X	3
Total	2	3	2	4	3	4	2	4	3	3	

Já a *suite* composta pelos testes 8, 2, 10, 3, 1 e 5 teria o seu valor de APFD definido por:

$$APFD = 1 - \frac{4+3+1+1+3+2+1+1+2+1}{10*10} + \frac{1}{2*10} = 0,86 \qquad (4.5)$$

Observa-se então que a taxa de detecção de falhas da *suite* 2 é bastante superior à da *suite* 1. Isto fica mais evidente quando se observa graficamente a cobertura de falhas para ambas as *suites*, à medida em que os testes vão sendo executados (Figura 9).

Figura 9 – Percentual de cobertura ao longo da execução das *suites* de teste.

A segunda métrica empregada na avaliação das soluções foi a $APFD_C$ (ELBAUM; MALISHEVSKY; ROTHERMEL, 2001), que assim como o APFD serve para mensurar a taxa de detecção de falhas de uma *suite* de testes, mas também considera o custo - geralmente o tempo de execução dos casos de teste - e a severidade das falhas. O $APFD_C$ de uma *suite* de testes é definido por:

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \qquad (4.6)$$

Onde:

- f_i - severidade da falha i
- t_j - custo do teste j (tempo de execução)

Vale observar que caso os custos ou severidades das falhas sejam idênticos, a fórmula continua sendo aplicável. Caso todos os custos e todas as severidades sejam idênticos, a fórmula do APFD_C se reduz à do APFD.

5 Resultados e Discussão

5.1 Objetos dos experimentos

Em nosso trabalho, foram empregados programas oriundos do SIR (*Software-Artifact Infrastructure Repository*[1]) um repositório público de objetos gratuitos para experimentação em engenharia de software (DO; ELBAUM; ROTHERMEL, 2005). Ele foi utilizado por conter objetos bastante empregados em outros trabalhos relacionados a teste de regressão, como (ELBAUM; MALISHEVSKY; ROTHERMEL, 2000), (HAO et al., 2016), (ZHENG et al., 2016), (JIANG; CHAN, 2015), dentre outros.

Foram escolhidos oito programas escritos na linguagem C (KERNIGHAN, 1988). O programa *Space* foi desenvolvido pela agência espacial europeia, e os demais foram desenvolvidos por pesquisadores da *Siemens Corporate Research* (HUTCHINS et al., 1994).

Os programas objeto contam com falhas semeadas pelos desenvolvedores, possibilitando a realização de experimentos relacionados a teste de software. Cada programa objeto dispõe de um *pool* de casos de teste, que envolve tanto testes de caixa-preta e caixa-branca (PRESSMAN, 2015). O pool original de testes é utilizado para criar 1000 *suites* de teste para cada programa, visando contemplar a maior cobertura de código da maneira mais realista possível. A experimentação empregou uma *suite* de casos de teste de cada objeto, escolhida aleatoriamente.

A Tabela 5 descreve os objetos empregados na experimentação, com suas quantidades de linhas, o número de componentes (funções em linguagem C), o total de casos de teste do objeto, o número médio de casos de teste por *suite* gerada a partir do pool inicial, e o número de casos de teste contidos na *suite* empregada na experimentação.

Tabela 5 – Objetos utilizados na experimentação.

Programa	**Finalidade**	**Linhas**	**Funções**	**Total de casos (*pool*)**	**Média de casos / *suite***	***Suite* utilizada (casos)**
print_tokens	analisador léxico	402	18	4130	16	20
print_tokens2	analisador léxico	483	21	4115	12	13
replace	Reconhecimento de padrões e substituição	516	22	5542	19	19
schedule	Agendador de prioridades	299	18	2650	8	11
schedule2	Agendador de prioridades	297	17	2710	8	10
space	Prevenção de colisão de aeronaves	6218	136	13585	155	155
tcas	Interpretador para uma linguagem de definição de arrays	138	9	1608	6	6
tot_info	programa de estatística	346	9	1052	7	5

[1] sir.unl.edu/

5.2 Implementação da abordagem

Para a experimentação, a abordagem foi implementada na linguagem Java (GOSLING et al., 2014). Em relação à obtenção das entradas da primeira etapa, a complexidade ciclomática dos objetos em linguagem C foi calculada utilizando o programa CCCC (*C and C++ Code Counter*)[2], e os acoplamentos aferente e eferente foram obtidos pela análise do relatório da árvore de invocações das funções do programa gerado pelo software *Scitools Understand*[3].

O sistema de inferência *fuzzy* foi implementado utilizando a biblioteca *jFuzzyLogic*[4] (CINGOLANI; ALCALÁ-FDEZ, 2013). O método de ativação (operador *fuzzy* de implicação, por meio do qual o sistema *fuzzy* realiza a inferência dos conjuntos *fuzzy* de saída com base nas proposições iniciais) empregado foi o *MIN*. O método de acumulação (operador de agregação, pelo qual o sistema obtém o grau de ativação de cada regra por meio de um operador conjuntivo) empregado foi o *MAX*. O método de defuzzificação escolhido foi o *Center of Area* (ROSS, 2004).

Na segunda etapa, para calcular a criticidade dos casos de teste, é necessário conhecer a cobertura dos casos de teste. Nos objetos do SIR, essa informação foi obtida utilizando o *gcov*, uma ferramenta que trabalha em conjunto com o *gcc*[5], um compilador da linguagem C. O *gcov* retorna o percentual de declarações executadas em cada função por cada caso de teste.

Na terceira etapa, o algoritmo *Max-Min Ant System* empregou na exploração do espaço de busca 80 agentes (formigas) por iteração, em um total de 6000 iterações. Foi atribuído o valor 1.0 para a importância do feromônio (α) e para a importância da informação heurística (β), e uma taxa de evaporação do feromônio de 5%.

Na avaliação dos resultados, a métrica APFD_C emprega o custo do caso de teste, que neste trabalho equivale ao tempo de execução e foi obtido do *log* gerado durante a execução dos casos de teste. Os objetos não contam com informação de severidade das falhas, informação que recebeu valor 1 para todas as falhas.

5.3 Definição de parâmetros de funcionamento da abordagem

Foi realizada uma experimentação inicial para definir alguns parâmetros de funcionamento do algoritmo *Max-Min Ant System* empregado na etapa de priorização. Foram avaliadas diversas configurações, onde foram alterados o número de agentes, o número de

[2] https://sourceforge.net/projects/cccc/
[3] https://scitools.com/
[4] jfuzzylogic.sourceforge.net/
[5] gcc.gnu.org

iterações do algoritmo, a importância das informações heurística e feromonal, e a função heurística empregada na construção das soluções.

Na experimentação inicial, a abordagem foi executada empregando diversas configurações, 100 repetições para cada configuração, e a melhor solução obtida em cada repetição foi registrada. Foram empregadas duas métricas de avaliação: o APFD e o $APFD_C$. O objeto empregado para essa etapa da experimentação foi o programa *space*, por ser o objeto que possui as quantidades mais expressivas de linhas de código, casos de teste e falhas semeadas. A heurística empregada pelo algoritmo MMAS para escolher os testes durante a construção das soluções, exceto quando explicitamente indicado o contrário, é representada pela fórmula:

$$\frac{TC_j * F_j}{T_j} \tag{5.1}$$

Onde:

- TC_j: criticidade do caso de teste j;
- T_j: tempo de execução do caso de teste j;
- F_j: quantidade de falhas detectadas pelo caso de teste j.

A heurística descrita acima foi escolhida por contemplar a importância de execução do caso de teste (criticidade), o potencial de detecção de falhas do caso de teste (com base no histórico de execução) e o custo de execução do caso de teste (tempo).

A normalidade dos dados foi testada empregando o teste de Shapiro-Wilk (SHAPIRO; WILK, 1965), e os resultados mostraram que alguns dados não seguiam uma distribuição normal (nível de significância = 0.01). Então a existência de diferença estatística foi calculada empregando um teste não-paramétrico (*Kruskal-Wallis* (KRUSKAL; WALLIS, 1952), nível de significância = 0.05). Nas subseções a seguir, serão descritos os resultados obtidos na experimentação inicial.

Quantidade de agentes

Para chegar a quantidade de agentes a ser empregada na avaliação estatística, a abordagem foi executada empregando diversas variações na quantidade de agentes empregados por iteração do algoritmo, com um número fixo de iterações (3000). As Tabelas 6 e 7 apresentam o resultado da comparação estatística entre as diferentes quantidades de agentes aplicadas. Em cada tabela, as células contém o *p-value* que representa a diferença estatística encontrada nos resultados empregando os números de agentes declarados no topo da coluna e no início da linha. Por exemplo, na Tabela 6, pode ser observado que, na

célula contida na linha de 20 agentes e na coluna de 10 agentes possui um *p-value* < 0.001. Isso indica que, a um nível de significância de 0.05, os resultados obtidos ao empregar 20 agentes são diferentes a nível estatístico dos resultados obtidos empregando 10 agentes. Isso também pode ser observado na comparação entre o emprego de 60 e 40 agentes, que retorna um *p-value* de 0.002, indicando que ainda existe diferença estatisticamente significativa entre os resultados. Ao comparar o emprego de 50 e 40 agentes, já é possível observar um *p-value* (0.288) maior que o grau de significância empregado (0.05), o que indica que não foi detectada diferença estatisticamente significativa entre os resultados. Isso também pode ser observado ao comparar o emprego de 70 e 80 agentes.

Tabela 6 – Diferença estatística (*p-values*) entre os resultados empregando diferentes quantidades de agentes e a métrica APFD

	10 ag.	**20 ag.**	**30 ag.**	**40 ag.**	**50 ag.**	**60 ag.**	**70 ag.**	**80 ag.**	**90 ag.**
20 ag.	<0.001	-	-	-	-	-	-	-	-
30 ag.	<0.001	<0.001	-	-	-	-	-	-	-
40 ag.	<0.001	<0.001	1	-	-	-	-	-	-
50 ag.	<0.001	<0.001	1	0.288	-	-	-	-	-
60 ag.	<0.001	<0.001	0.052	0.002	1	-	-	-	-
70 ag.	<0.001	<0.001	0.003	<0.001	0.521	1	-	-	-
80 ag.	<0.001	<0.001	<0.001	<0.001	0.019	1	1	-	-
90 ag.	<0.001	<0.001	<0.001	<0.001	<0.001	0.141	1	1	-
100 ag.	<0.001	<0.001	<0.001	<0.001	<0.001	0.0003	0.035	0.694	1

Tabela 7 – Diferença estatística (*p-values*) entre os resultados empregando diferentes quantidades de agentes e a métrica APFD_C

	10 ag.	**20 ag.**	**30 ag.**	**40 ag.**	**50 ag.**	**60 ag.**	**70 ag.**	**80 ag.**	**90 ag.**
20 ag.	1	-	-	-	-	-	-	-	-
30 ag.	<0.001	0.215	-	-	-	-	-	-	-
40 ag.	<0.001	<0.001	1	-	-	-	-	-	-
50 ag.	<0.001	<0.001	0.014	1	-	-	-	-	-
60 ag.	<0.001	<0.001	<0.001	0.019	0.618	-	-	-	-
70 ag.	<0.001	<0.001	<0.001	0.011	0.349	1	-	-	-
80 ag.	<0.001	<0.001	<0.001	<0.001	0.015	1	1	-	-
90 ag.	<0.001	<0.001	<0.001	<0.001	<0.001	0.505	1	1	-
100 ag.	0.021	1	1	0.011	<0.001	<0.001	<0.001	<0.001	<0.001

A partir de 10 agentes, foram experimentadas diversas configurações onde variavam as quantidades de agentes, sempre sob um incremento de 10 unidades. Cada configuração foi executada 100 vezes. Para a métrica APFD, o acréscimo de agentes até o número de 30 sempre culminou em aumento, com diferença estatisticamente significante no resultado obtido. A partir de 40 agentes, a diferença nos resultados advinda do acréscimo de agentes passou a não ser estatisticamente significante em relação à quantidade imediatamente anterior, mas ainda bastante relevante quando comparado às demais quantidades anteriores. Para a métrica APFD_C, o ganho obtido com o aumento de agentes foi ainda menor, sendo que desde o uso de 20 agentes a diferença já não era estatisticamente significante.

Com base no crescimento dos resultados para ambas as métricas nas diferentes configurações, nas próximas etapas da experimentação inicial serão utilizados 80 agentes por rodada, onde o ganho em termos de resultados passou a ser ainda menor em relação às quantidades anteriores.

Quantidade de iterações

Para definir a quantidade de iterações do algoritmo para a experimentação, a abordagem foi executada empregando 10 configurações diferentes, 100 repetições para cada configuração. Em cada configuração, foram empregados 80 agentes e uma quantidade variável de iterações no algoritmo ACO. A análise estatística dos resultados está representada nas Tabelas 8 e 9.

Tabela 8 – Diferença estatística entre os resultados para a métrica APFD empregando diferentes quantidades de iterações no algoritmo MMAS

	1000	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**
2000	0.030	-	-	-	-	-	-	-	-
3000	<0.001	0.045	-	-	-	-	-	-	-
4000	<0.001	<0.001	1	-	-	-	-	-	-
5000	<0.001	<0.001	0.058	1	-	-	-	-	-
6000	<0.001	<0.001	<0.001	0.092	1	-	-	-	-
7000	<0.001	<0.001	<0.001	0.053	1	1	-	-	-
8000	<0.001	<0.001	<0.001	<0.001	0.046	1	1	-	-
9000	<0.001	<0.001	<0.001	<0.001	0.000	0.599	0.533	1	-
10000	<0.001	<0.001	<0.001	0.104	1	1	1	1	0.508

Tabela 9 – Diferença estatística entre os resultados para a métrica APFD_C empregando diferentes quantidades de iterações no algoritmo MMAS

	1000	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**	**8000**	**9000**
2000	<0.001	-	-	-	-	-	-	-	-
3000	<0.001	1	-	-	-	-	-	-	-
4000	<0.001	0.002	0.002	-	-	-	-	-	-
5000	<0.001	0.002	0.001	1	-	-	-	-	-
6000	<0.001	<0.001	<0.001	0.222	0.693	-	-	-	-
7000	<0.001	<0.001	<0.001	0.014	0.091	1	-	-	-
8000	<0.001	<0.001	<0.001	<0.001	<0.001	1	1	-	-
9000	<0.001	<0.001	<0.001	<0.001	0.006	1	1	1	-
10000	<0.001	<0.001	<0.001	<0.001	<0.001	1	1	1	1

Para ambas as métricas, a partir de 4000 iterações, o acréscimo no número de iterações passou a proporcionar melhorias cada vez menos significativas aos resultados. Com base na análise dos resultados, optou-se por empregar 6000 iterações nas próximas etapas do experimento, quantidade onde não foi observado ganho com diferença estatisticamente significante em comparação às duas quantidades anteriores (4000 e 5000 iterações).

Importância da informação feromonal

A pré-experimentação também teve por objetivo avaliar a importância da informação feromonal na etapa de priorização. Para isso, os resultados foram comparados aos obtidos com uma variação da abordagem cujo algoritmo MMAS da terceira etapa foi executado sem considerar a informação feromonal. Os resultados (Tabela 10) evidenciaram que a não-utilização da informação feromonal acarretou uma diminuição nos resultados para ambas as métricas, com diferença estatisticamente significante ($p\text{-}value < 0.001$) .

Tabela 10 – Avaliação da importância da informação feromonal

	APFD		$APFD_C$	
	com ferom.	sem ferom.	com ferom.	sem ferom.
Média	96.86	95.15	97.05	95.65
Mediana	96.82	95.08	97.03	95.58
Desvio	0.004	0.003	0.002	0.003
Kruskal-Wallis	p-value < 0.001		p-value < 0.001	

Importância da informação Heurística

A avaliação da importância da informação heurística foi realizada de maneira similar à avaliação da importância da informação feromonal. Os resultados obtidos com a abordagem empregando todas as informações foram comparados aos valores obtidos empregando somente a informação feromonal. Os resultados são apresentados na Tabela 11.

Tabela 11 – Avaliação da importância da informação heurística

	APFD		$APFD_C$	
	com heur.	sem heur.	com heur.	sem heur.
Média	96.86	95.87	97.05	95.96
Mediana	96.82	95.84	97.03	95.93
Desvio	0.004	0.003	0.003	0.004
Kruskal-Wallis	p-value < 0.001		p-value < 0.001	

Assim como a informação feromonal, a informação heurística se mostrou importante dentro do algoritmo, e sua remoção representou uma diminuição nos resultados obtidos para ambas as métricas, com diferença estatisticamente significante.

Avaliação de heurísticas

Na última etapa da pré-experimentação, foram avaliadas algumas opções de heurísticas de seleção dos casos de teste para construção das soluções do algoritmo ACO. Os resultados da avaliação estão nas Tabelas 12 e 13.

- Heurística 1 (H_1): TC_j/T_j
- Heurística 2 (H_2): $(TC_j * F_j)/T_j$
- Heurística 3 (H_3): F_j
- Heurística 4 (H_4): $TC_j * F_j$
- Heurística 5 (H_5): TC_j

Tabela 12 – Resultados para a métrica APFD (a) obtidos com a utilização das heurísticas, e (b) análise da diferença estatística entre os resultados

(a)

	H_1	H_2	H_3	H_4	H_5
Média	95.36	96.86	97.01	96.87	95.26
Mediana	95.35	96.82	96.95	96.84	95.27
Desvio	0.004	0.002	0.001	0.002	0.003

(b)

	H_1	H_2	H_3	H_4
H_2	<0.001	-	-	-
H_3	<0.001	<0.001	-	-
H_4	<0.001	1	1	-
H_5	1	<0.001	<0.001	<0.001

Tabela 13 – Resultados para a métrica $APFD_C$ (a) obtidos com a utilização das heurísticas, e (b) análise da diferença estatística entre os resultados

(a)

	H1	H2	H3	H4	H5
Média	95.84	97.05	97.12	97.07	95.35
Mediana	95.82	97.03	97.14	97.05	95.32
Desvio	0.003	0.002	0.002	0.002	0.003

(b)

	H1	H2	H3	H4
H2	<0.001	-	-	-
H3	<0.001	0.939	-	-
H4	<0.001	1	<0.001	-
H5	<0.001	<0.001	<0.001	<0.001

Com base nos resultados, é possível perceber que o emprego do histórico de falhas nas heurísticas repercutiu positivamente no resultado final, com as heurísticas 2, 3 e 4 se destacando a níveis estatisticamente significantes. Conforme esperado, o emprego do tempo no cálculo da informação heurística não apresentou efeitos significativos nos resultados para a métrica APFD. Isso fica evidenciado pela ausência de diferença estatisticamente significante na comparação entre as heurísticas 2 e 4, ou entre as heurísticas 1 e 5.

Ainda dentro das expectativas, a heurística 1 se mostrou significativamente superior à heurística 5 quando foi avaliada sob a métrica $APFD_C$, evidenciando a importância de empregar o custo (o tempo) ao priorizar os testes buscando maximizar essa métrica. Os resultados obtidos com as heurísticas 2 e 4, apesar de não apresentarem diferença estatisticamente significante nos resultados mesmo sob a variação do tempo, também não tiveram diferença estatisticamente significante dos da heurística 3, que continuou a se comportar melhor mas sob uma margem bastante estreita. Isso pode indicar talvez a proximidade dos resultados a um valor ótimo global da métrica para o objeto.

O emprego do histórico de falhas dos casos de teste trouxe um ganho estatisticamente significante para os resultados. Porém, o fato de um caso de testes ter encontrado falha em

uma versão anterior não significa necessariamente que ele continuará a encontrar falhas. É interessante então empregar uma heurística que considere outras informações, motivo pelo qual resolveu-se empregar na experimentação a heurística 2, que considera também a criticidade e o tempo de execução dos casos de teste.

Para ambas as métricas de avaliação, os melhores resultados foram obtidos ao empregar a heurística 3, que aplicava puramente a informação do histórico de falhas. Isso era esperado, visto que a métrica APFD é ligada precisamente à ocorrência de falhas. Hao et al. (2016) também implementaram uma técnica de priorização que ordenava a execução dos testes com base no histórico de detecção de falhas, que apesar de não ter aplicabilidade prática servia como técnica de controle. Então, a heurística 3 mostra-se bastante interessante, não como sendo utilizável na abordagem, mas ao servir como patamar para a análise dos resultados. Sob esse prisma, vale a pena destacar que, para a métrica APFD_C, os resultados obtidos com a heurística 2 não tiveram diferença estatística significante para os obtidos com a heurística 3.

5.4 Experimentação

Para a condução da avaliação da proposta, a abordagem foi executada então 1000 vezes para cada um dos objetos de experimentação descritos na Subseção 5.1. Com base nos resultados da experimentação inicial, o algoritmo de priorização (MMAS) empregou 80 agentes, 6000 iterações e a heurística 2 ($(TC_j * F_j)/T_j$.)

Para fins de controle, foram obtidos os valores do APFD do conjunto de testes seguindo:

- a ordenação original dos casos de teste (ordem na qual os casos de teste foram listados, na *suite* escolhida do objeto de experimentação);
- a ordenação empregando uma técnica gulosa, que pontua os casos de teste segundo a heurística empregada no algoritmo de priorização ($(TC_j * F_j)/T_j$); e
- a ordenação ótima, encontrada por meio de uma busca exaustiva (quando possível).

Além desses valores, a abordagem foi submetida a um teste de sanidade, um teste básico que contrasta os resultados obtidos com a abordagem com os obtidos empregando outras técnicas, e teve seus resultados comparados em termos de significância estatística e prática com os obtidos por uma busca aleatória, também executada 1000 vezes para cada objeto. Os dados novamente apresentaram distribuições não-normais, demandando mais uma vez a execução de testes não-paramétricos. A diferença estatística entre os resultados da abordagem e os da busca aleatória foi então obtida por meio de um teste de Wilcoxon–Mann–Whitney (RUMSEY, 2007) com correção de Bonferroni (DUNN, 1959).

O teste de significância estatística pode ser complementado por um teste de significância prática, que observa a magnitude da melhoria da abordagem proposta em relação à busca aleatória. Para isso, foi empregada a medida A de Vargha-Delaney (VARGHA; DELANEY, 2000), uma métrica de análise de tamanho do efeito.

Os valores para essa métrica variam entre 0 e 1, sendo que quanto mais perto de 0.5, mais similares são os grupos sendo comparados. Na análise realizada, valores superiores a 0.5 indicam que a abordagem proposta apresentou resultados melhores do que a busca aleatória. As Tabelas 14 e 15 posicionam os resultados da abordagem frente aos demais resultados obtidos.

Tabela 14 – Comparação entre os resultados obtidos com a abordagem proposta, os conjuntos originais (não ordenados) de testes, uma ordenação obtida por meio de um algoritmo guloso e a ordenação aleatória.

	Controle			**Experimento**				
	Orig.	**Guloso**	**Ótimo**	**Aleat.**	**Proposta**	**Desvio (Proposta)**	**Dif. Estat.**	**Dif. Prática (Medida *A*)**
print_tokens	82.50	96.42	-	76.12	96.42	-	<0.001	0.857
print_tokens2	73.07	88.46	96.15	68.17	96.15	-	<0.001	0.923
replace	83.47	83.87	-	49.84	97.36	-	<0.001	0.974
schedule	54.54	63.63	95.45	59.83	95.45	-	<0.001	0.958
schedule2	24.99	35.00	95.00	49.07	95.00	-	<0.001	1
tcas	54.62	71.29	84.25	51.30	84.25	-	<0.001	0.996
totinfo	67.14	67.14	81.42	54.55	81.42	-	<0.001	0.895
space	83.47	83.87	-	83.07	95.62	0.34	<0.001	1

Tabela 15 – Comparação entre os resultados obtidos com a abordagem proposta, os conjuntos originais (não ordenados) de testes, uma ordenação obtida por meio de um algoritmo guloso e a ordenação aleatória - APFD_C.

	Controle			**Experimento**				
	Orig.	**Guloso**	**Ótimo**	**Aleat.**	**Proposta**	**Desvio (Proposta)**	**Dif. Estat.**	**Dif. Prática (Medida *A*)**
print_tokens	82.5	96.66	-	76.12	96.66	-	<0.001	0.857
print_tokens2	73.07	88.46	96.15	68.17	96.15	-	<0.001	0.923
replace	52.77	63.88	-	49.89	97.22	-	<0.001	0.974
schedule	54.54	63.63	96.01	59.88	96.01	-	<0.001	0.958
schedule2	16.66	38.88	94.44	49.07	94.44	-	<0.001	1
tcas	59.09	77.27	83.33	51.36	83.33	-	<0.001	0.9965
totinfo	68.36	68.36	84.19	54.64	84.19	-	<0.001	0.895
space	83.39	83.87	-	83.07	95.76	0.003	<0.001	1

Analisando os valores de controle (APFD das ordenações original, gulosa e ótima) pôde ser percebido que, exceto para o objeto *totinfo*, o algoritmo guloso gerou avaliações melhores do que a não-ordenação para ambas as métricas, apesar de ter seus resultados situados a uma distância considerável do valor ótimo obtido. A média dos resultados obtidos com a abordagem se mostrou igual ou superior ao valor obtido com as ordenações de controle, para todos os objetos e em ambas as métricas. Para todos os objetos onde foi possível realizar uma busca exaustiva, os resultados da abordagem alcançaram o valor ótimo global. Quanto ao teste de sanidade, quando comparada à busca aleatória a abordagem proposta apresentou resultados superiores a níveis estatísticos e práticos

(tamanho do efeito). Os resultados evidenciam, então, a aplicabilidade da abordagem à priorização de casos de teste.

6 Conclusões e Trabalhos Futuros

Visando melhorar a realização do teste de regressão, este trabalho propôs a priorização de casos de teste por meio de uma abordagem que busca maximizar a taxa de detecção de falhas das *suites* de teste considerando métricas extraídas dos componentes do software.

A abordagem foi gerada com base na proposta em (SILVA et al., 2016), que mesclava a seleção e a priorização de casos de teste. As modificações em relação a esse trabalho deram-se especialmente nas etapas iniciais e finais da abordagem. As duas etapas iniciais - cálculo da relevância das classes e inferência da criticidade das classes - objetivavam a obtenção da criticidade (importância de verificação) do componente de software e foram abstraídas, tornando-se a etapa de cálculo da criticidade dos componentes de software. Essa modificação teve o intuito de tornar a abordagem mais amigável a outros paradigmas de programação - visto que a experimentação empregou objetos feitos em linguagem C - mas não impede que, no caso de utilização da abordagem proposta em projetos baseados em programação orientada a objetos, a criticidade seja obtida com base nas duas primeiras etapas da abordagem original. A criticidade dos componentes considera métricas de software relacionadas à susceptibilidade a falhas.

A etapa de cálculo da criticidade dos casos de teste continua sendo essencialmente baseada na cobertura dos casos, e na criticidade dos componentes por eles cobertos. É válido observar que a cobertura dos casos de teste já foi empregada em outros trabalhos referentes à melhoria do teste de regressão, mas durante a realização deste trabalho não foi identificada nenhuma pesquisa que considerasse características particulares dos componentes cobertos.

A etapa de seleção do trabalho original foi suprimida, em prol de um melhor aproveitamento do potencial total de detecção de falhas da *suite* de testes. Já a etapa de priorização ganhou maior relevância, passando a considerar a taxa de detecção de falhas da *suite* de testes - indicativo de qualidade da solução mais comumente identificado em trabalhos relacionados ao teste de regressão - ao invés de fazer uma ordenação tomando a criticidade dos casos de teste como chave. Portanto, ao invés de aplicar uma restrição de tempo à solução, optou-se por ordenar todo o conjunto de testes, que será executado conforme a disponibilidade de recursos da equipe.

A heurística empregada na construção das soluções considera a criticidade dos casos de teste, em conjunto com os seus tempos de execução e históricos de detecção de falhas. Desta forma, as ordenações geradas consideram a importância de se executar (na forma da criticidade), a capacidade 'comprovada' de detecção de falhas (por meio do

histórico de falhas) e o tempo de execução (custo) dos casos de teste.

Seguindo as recomendações de (CATAL; MISHRA, 2013), foram empregadas métricas conhecidas de avaliação dos resultados, e a abordagem empregou as duas métricas conhecidas de mensuração de taxa de detecção de falhas: o APFD e o $APFD_C$. Ainda sobre esse trabalho, outra recomendação nele contida é a utilização de objetos oriundos de um repositório público. Isso deve facilitar a reprodutibilidade dos experimentos e ajudar, em trabalhos futuros, a melhor posicionar os resultados da abordagem em relação a outros trabalhos.

Na experimentação, as *suites* de testes obtidas com a abordagem foram comparadas inicialmente com a ordenação original das *suites*; com uma ordenação obtida por meio de um algoritmo guloso, que considerou criticidade, tempo de execução e histórico de falhas dos casos de teste; e com os resultados de uma busca exaustiva. A média dos resultados indicou, para ambas as métricas de avaliação, uma melhoria na taxa de detecção de falhas em comparação à *suite* não ordenada (ordem original). Isso se repetiu para a comparação com o resultado obtido com o algoritmo guloso. Para todos os programas objeto de experimentação onde foi possível a execução da busca exaustiva, a abordagem alcançpu, em todas as repetições do experimento em cada objeto, o resultado ótimo.

Para uma análise estatística mais aprofundada, foi realizado um teste de sanidade (*sanity check*), onde os resultados obtidos com a abordagem foram comparados aos resultados obtidos com um algoritmo de busca aleatória. O teste indicou que a abordagem se mostrou superior, em termos de significância estatística e prática (tamanho do efeito), aos resultados da busca aleatória. Os resultados acima descritos evidenciam a aplicabilidade e eficácia da abordagem a ambientes de testes de regressão.

6.1 Limitações

Apesar dos resultados alcançados, algumas limitações a respeito do trabalho devem ser apontadas. Os objetos empregados na experimentação são bastante empregados em trabalhos relacionados ao teste de regressão (HAO et al., 2016; ZHENG et al., 2016; PANICHELLA et al., 2015; ELBAUM; MALISHEVSKY; ROTHERMEL, 2000; ROTHERMEL et al., 2001), e consistem de programas reais. O objeto *space* conta com falhas reais, reportadas durante seu desenvolvimento. As falhas dos demais objetos foram semeadas pelos desenvolvedores de maneira a oferecer algum grau de verossimilhança (HUTCHINS et al., 1994; DO; ELBAUM; ROTHERMEL, 2005). No entanto, os objetos ainda podem ser considerados pequenos (possuem poucas linhas de código e unidades estruturais, como funções) e pouco representativos da realidade de projetos reais de software.

A experimentação poderia ter se beneficiado da aplicação de um *dataset* extraído de um projeto real. No entanto, não foi possível obter esse projeto real em tempo hábil

à realização do trabalho. Buscou-se trabalhar com projetos disponíveis em repositórios públicos, como o *GitHub*[1] e o *bitbucket*[2]. No entanto, em todos os projetos estudados, deparou-se com alguma das seguintes dificuldades que impossibilitaram sua utilização:

- quantidade pouco significativa de casos de teste, ou tempo de execução total reduzido;
- ausência de falhas nas versões "commitadas"do programa no repositório;
- impossibilidade de mapear todas as informações necessárias (complexidade ciclomática, instabilidade) dos componentes do software ou dos casos de teste (cobertura, localização de falhas) por meio das ferramentas de instrumentação disponíveis atualmente.

Alguns elementos da abordagem original também não puderam ser considerados neste trabalho:

- Nos objetos de experimentação, não foi possível obter nenhum tipo de informação referente a alterações no código, em relação a versões anteriores do programa. Esse era um critério empregado na etapa de seleção dos casos de teste. Apesar de essa etapa ter sido suprimida da versão atual da abordagem, poderia ser interessante considerar essa informação no cálculo da criticidade dos componentes de software. A localização das falhas dos componentes pode ser mapeada ao comparar o código original e os códigos onde foram semeadas as falhas. No entanto, não se pode considerar como modificados somente os trechos do programa onde ocorreram falhas.
- A relevância das *features* do sistema não foi mapeada para os componentes do software. A dificuldade em mapear *features* em um código com o qual não se seja familiar já foi abordada em (EISENBERG; VOLDER, 2005). Entretanto, durante a execução deste trabalho não foi possível atribuir graus de relevância às *features* dos objetos empregados na experimentação. No entanto, a aplicação da abordagem em objetos com os quais se seja familiar (por exemplo, um sistema que esteja sendo desenvolvido pelo usuário da abordagem) pode empregar as etapas iniciais da abordagem de (SILVA et al., 2016) para realizar o cálculo da criticidade dos casos de teste.

Infelizmente, não foi possível obter a informação da severidade das falhas dos objetos do repositório, tampouco inferi-la de outra maneira. A experimentação poderia ter se beneficiado da informação na avaliação dos resultados obtidos com a métrica $APFD_C$. Em uma aplicação prática da abordagem, poderia ser interessante aplicar a relevância

[1] https://github.com/
[2] https://bitbucket.org/

dos componentes de software em conjunto com a severidade das falhas, na obtenção da criticidade dos componentes de software.

6.2 Trabalhos Futuros

Com base no apresentado, alguns trabalhos podem ser propostos para a continuidade da pesquisa:

- Elaboração de uma base de dados mais representativa da realidade, registrando informações de um projeto real de grande porte, com suas várias versões e casos de teste.
- Elaboração de um repositório de objetos de experimentação, que além de armazenar os programas e suas diversas versões, armazene também os trabalhos que empregaram os programas, e as informações extraídas desses programas e empregadas nos trabalhos, facilitando a replicação de experimentos. Essa tarefa já está sendo realizada por um membro do laboratório OASIS[3].
- Realização de estudos mais aprofundados sobre a relação entre métricas de software e susceptibilidade a falhas, no intuito de aprimorar o cálculo da criticidade.
- Definição de uma métrica para avaliar a solução gerada com base na disposição da criticidade dos casos de teste ordenados.

[3] *Optimization, Autonomous Solutions and Intelligent Systems* LAB

Referências

ABRAN, A. et al. (Ed.). *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001. Citado na página 1.

ACHARYA, A. A.; KHANDAI, S.; MOHAPATRA, D. P. A novel approach for test case prioritization using business criticality test value. *International Journal of Computer Applications*, v. 46, n. 15, p. 1–8, May 2012. Citado 4 vezes nas páginas 5, 11, 19 e 23.

ASKARUNISA, M. A.; SHANMUGAPRIYA, M. L.; RAMARAJ, D. N. Cost and coverage metrics for measuring the effectiveness of test case prioritization techniques. *INFOCOMP Journal of Computer Science*, v. 9(1), p. 43–52, 2010. Citado 2 vezes nas páginas 1 e 4.

BEIZER, B. *Software System Testing and Quality Assurance*. New York, NY, USA: Van Nostrand Reinhold Co., 1984. ISBN 0-442-21306-9. Citado na página 9.

BLACK, J.; MELACHRINOUDIS, E.; KAELI, D. Bi-criteria models for all-uses test suite reduction. In: *Proceedings. 26th International Conference on Software Engineering.* [S.l.: s.n.], 2004. p. 106–115. ISSN 0270-5257. Citado na página 18.

BOURQUE, P.; FAIRLEY, R. E. (Ed.). *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014. Citado na página 1.

CARLSON, R.; DO, H.; DENTON, A. A clustering approach to improving test case prioritization: An industrial case study. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM).* [S.l.: s.n.], 2011. p. 382–391. ISSN 1063-6773. Citado na página 19.

CATAL, C.; MISHRA, D. Test case prioritization: A systematic mapping study. *Software Quality Journal*, Kluwer Academic Publishers, Hingham, MA, USA, v. 21, n. 3, p. 445–478, set. 2013. Citado 8 vezes nas páginas 2, 11, 12, 17, 20, 26, 33 e 50.

CINGOLANI, P.; ALCALÁ-FDEZ, J. jfuzzylogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming. *International Journal of Computational Intelligence Systems*, Taylor & Francis Group, v. 6, n. sup1, p. 61–75, 2013. Citado na página 40.

DO, H.; ELBAUM, S. G.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 2005. Citado 4 vezes nas páginas 4, 6, 39 e 50.

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, Sept 2010. Citado 3 vezes nas páginas 3, 9 e 12.

DORIGO, M.; BONABEAU, E.; THERAULAZ, G. Ant algorithms and stigmergy. *Future Generations Computer Systems*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 16, n. 9, p. 851–871, jun. 2000. Citado na página 13.

DORIGO, M.; STüTZLE, T. *An Experimental Study of the Simple Ant Colony Optimization Algorithm*. 2001. Citado na página 13.

DUNN, O. J. Estimation of the medians for dependent variables. *The Annals of Mathematical Statistics*, v. 30, n. 1, p. 192–197, 1959. Citado na página 46.

EISENBERG, A. D.; VOLDER, K. D. Dynamic feature traces: finding features in unfamiliar code. In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. [S.l.: s.n.], 2005. p. 337–346. ISSN 1063-6773. Citado na página 51.

ELBAUM, S.; GABLE, D.; ROTHERMEL, G. Understanding and measuring the sources of variation in the prioritization of regression test suites. In: *Proceedings of the 7th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 2001. (METRICS '01), p. 169–. Citado na página 17.

ELBAUM, S. et al. Understanding the effects of changes on the cost-effectiveness of regression testing techniques. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., v. 13, n. 2, p. 65–83, 2003. ISSN 1099-1689. Citado 2 vezes nas páginas 2 e 9.

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. Citado 2 vezes nas páginas 17 e 37.

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. Citado 6 vezes nas páginas 4, 12, 17, 36, 39 e 50.

ELBAUM, S.; MALISHEVSKY, A. G.; ROTHERMEL, G. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 28, n. 2, p. 159–182, fev. 2002. ISSN 0098-5589. Citado 2 vezes nas páginas 17 e 36.

ENGELBRECHT, A. P. *Computational Intelligence: An Introduction*. 2nd. ed. [S.l.]: Wiley Publishing, 2007. Citado na página 13.

GAO, D.; GUO, X.; ZHAO, L. Test case prioritization for regression testing based on ant colony optimization. In: *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. [S.l.: s.n.], 2015. p. 275–279. Citado na página 18.

GILL, N. S.; SIKKA, S. Fault proneness of classes in object-oriented systems. *International Journal of Advances in Embedeed System Research*, Jan 2011. Citado 4 vezes nas páginas 5, 20, 26 e 34.

GOSLING, J. et al. *The Java Language Specification, Java SE 8 Edition*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2014. ISBN 013390069X, 9780133900699. Citado na página 40.

HAAG, S.; RAJA, M. K.; SCHKADE, L. L. Quality function deployment usage in software development. *Communications of the ACM*, ACM, New York, NY, USA, v. 39, n. 1, p. 41–49, jan. 1996. Citado na página 24.

HAO, D. et al. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering*, v. 42, n. 5, p. 490–505, May 2016. ISSN 0098-5589. Citado 6 vezes nas páginas 17, 20, 33, 39, 46 e 50.

HARMAN, M. Making the case for morto: Multi objective regression test optimization. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. [S.l.: s.n.], 2011. p. 111–114. Citado 2 vezes nas páginas 18 e 20.

HETZEL, W. C.; HETZEL, B. *The Complete Guide to Software Testing*. 2nd. ed. New York, NY, USA: John Wiley & Sons, Inc., 1991. ISBN 0471565679. Citado na página 9.

HUTCHINS, M. et al. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: *Proceedings of the 16th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. (ICSE '94), p. 191–200. Citado 2 vezes nas páginas 39 e 50.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1–84,, Dec 1990. Citado 2 vezes nas páginas 2 e 9.

JEFFREY, D.; GUPTA, N. Experiments with test case prioritization using relevant slices. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 81, n. 2, p. 196–221, fev. 2008. Citado na página 18.

JIANG, B.; CHAN, W. Input-based adaptive randomized test case prioritization: A local beam search approach. *Journal of Systems and Software*, v. 105, p. 91 – 106, 2015. Citado na página 39.

KELLERER, H.; PFERSCHY, U.; PISINGER, D. *Knapsack Problems*. [S.l.]: Springer, 2004. ISBN 9783540402862. Citado na página 23.

KERNIGHAN, B. W. *The C Programming Language*. 2nd. ed. [S.l.]: Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709. Citado na página 39.

KHANDAI, S.; ACHARYA, A. A.; MOHAPATRA, D. P. Prioritizing test cases using business criticality test value. *International Journal of Advanced Computer Science and Applications*, v. 3, n. 5, 2011. Citado 3 vezes nas páginas 5, 19 e 23.

KIM, J.-M.; PORTER, A. A history-based test prioritization technique for regression testing in resource constrained environments. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002. (ICSE '02), p. 119–129. Citado 2 vezes nas páginas 11 e 18.

KIM, S.; BAIK, J. An effective fault aware test case prioritization by incorporating a fault localization technique. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, NY, USA: ACM, 2010. (ESEM '10), p. 5:1–5:10. ISBN 978-1-4503-0039-1. Citado na página 18.

KRUSKAL, W. H.; WALLIS, W. A. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, American Statistical Association, v. 47, n. 260, p. 583–621, 1952. Citado na página 41.

LI, S. et al. A simulation study on some search algorithms for regression test case prioritization. In: WANG, J.; CHAN, W. K.; KUO, F.-C. (Ed.). *2010 10th International Conference on Quality Software*. [S.l.]: IEEE Computer Society, 2010. p. 72–81. Citado na página 17.

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, April 2007. Citado na página 18.

LU, Y. et al. How does regression test prioritization perform in real-world software evolution? In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 535–546. ISBN 978-1-4503-3900-1. Citado na página 2.

MALISHEVSKY, A. G.; ROTHERMEL, G.; ELBAUM, S. Modeling the cost-benefits tradeoffs for regression testing techniques. In: *International Conference on Software Maintenance, 2002. Proceedings.* [S.l.: s.n.], 2002. p. 204–213. ISSN 1063-6773. Citado na página 17.

MAMDANI, E. H. Application of fuzzy logic to approximate reasoning using linguistic synthesis. *IEEE Transactions on Computers*, C-26, n. 12, p. 1182–1191, Dec 1977. Citado na página 25.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. Citado 2 vezes nas páginas 3 e 34.

MCCABE, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, Dec 1976. Citado 3 vezes nas páginas 3, 25 e 34.

MCNEILL, F. M.; THRO, E. *Fuzzy Logic: A Practical Approach*. San Diego, CA, USA: Academic Press Professional, Inc., 1994. Citado na página 13.

NARCISO, E. N.; DELAMARO, M. E.; NUNES, F. de Lourdes dos S. Test case selection: A systematic literature review. *International Journal of Software Engineering and Knowledge Engineering*, v. 24, n. 4, p. 653–676, 2014. Citado na página 10.

PANICHELLA, A. et al. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, v. 41, n. 4, p. 358–383, April 2015. Citado 3 vezes nas páginas 11, 12 e 50.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. 8. ed. New York, NY, USA: McGrawHill, Inc., 2015. Citado 2 vezes nas páginas 1 e 39.

QU, B. et al. Test case prioritization for black box testing. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2007. (COMPSAC '07), p. 465–474. ISBN 0-7695-2870-8. Citado na página 19.

REINELT, G. *The traveling salesman: computational solutions for TSP applications*. [S.l.]: Springer-Verlag, 1994. Citado na página 35.

ROSS, T. J. *Fuzzy Logic with Engineering Applications*. [S.l.]: John Wiley & Sons, 2004. Citado na página 40.

ROTHERMEL, G. et al. The impact of test suite granularity on the cost-effectiveness of regression testing. In: *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002. (ICSE '02), p. 130–140. Citado na página 17.

ROTHERMEL, G.; HARROLD, M. J. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, v. 22, n. 8, p. 529–551, Aug 1996. Citado 2 vezes nas páginas 2 e 10.

ROTHERMEL, G. et al. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., v. 12, n. 4, p. 219–249, 2002. Citado 2 vezes nas páginas 3 e 10.

ROTHERMEL, G. et al. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, v. 27, n. 10, p. 929–948, Oct 2001. Citado 3 vezes nas páginas 5, 17 e 50.

ROTHERMEL, G. et al. Test case prioritization: an empirical study. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. [S.l.: s.n.], 1999. p. 179–188. Citado 3 vezes nas páginas 2, 9 e 17.

RUMSEY, D. *Intermediate Statistics For Dummies*. [S.l.]: Wiley, 2007. (–For dummies). ISBN 9780470147740. Citado na página 46.

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). *Biometrika*, v. 52, n. 3/4, p. 591–611, Dec. 1965. Citado na página 41.

SILVA, D. et al. A hybrid approach for test case prioritization and selection. In: *IEEE Congress on Evolutionary Computation (CEC)*. [S.l.: s.n.], 2016. Citado 10 vezes nas páginas 15, 4, 6, 7, 23, 24, 33, 34, 49 e 51.

SINGH, Y.; KAUR, A.; SURI, B. Test case prioritization using ant colony optimization. *SIGSOFT Software Engineering Notes*, ACM, New York, NY, USA, 2010. Citado na página 20.

SOMMERVILLE, I. *Software Engineering*. 10. ed. [S.l.]: Pearson, 2015. Hardcover. Citado na página 1.

STÜTZLE, T.; HOOS, H. H. Max-min ant system. *Future generation computer systems*, Elsevier, v. 16, n. 8, p. 889–914, 2000. Citado na página 27.

SURI, B.; SINGHAL, S. Understanding the effect of time-constraint bounded novel technique for regression test selection and prioritization. *International Journal of System Assurance Engineering and Management*, Springer India, v. 6, n. 1, p. 71–77, 2015. Citado na página 20.

TRICENTIS. *Software Fail Watch: 2016 in Review*. [S.l.]: Relatório técnico, 2017. Citado na página 1.

VARGHA, A.; DELANEY, H. D. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, v. 25, n. 2, p. 101–132, 2000. Citado na página 47.

WONG, W. E. et al. Effect of test set minimization on fault detection effectiveness. In: *Proceedings of the 17th International Conference on Software Engineering*. New York, NY, USA: ACM, 1995. (ICSE '95), p. 41–50. Citado na página 12.

YOO, S.; HARMAN, M. Pareto efficient multi-objective test case selection. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2007. (ISSTA '07), p. 140–150. Citado 4 vezes nas páginas 2, 18, 20 e 33.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd, v. 22, n. 2, p. 67–120, 2010. Citado 5 vezes nas páginas 2, 3, 9, 12 e 17.

ZADEH, L. A. Fuzzy sets. *Information and Control*, v. 8, p. 338–353, 1965. Citado na página 13.

ZADEH, L. A. Fuzzy logic = computing with words. *IEEE Transactions on Fuzzy Systems*, v. 4, n. 2, p. 103–111, May 1996. Citado 3 vezes nas páginas 3, 13 e 14.

ZHENG, W. et al. Multi-objective optimisation for regression testing. *Information Sciences*, v. 334, p. 1 – 16, 2016. Citado 2 vezes nas páginas 39 e 50.