Universidade Federal de Juiz de Fora
Campus Juiz de Fora
Bacharelado em Ciência da Computação

Estudo e aplicação de técnicas de teste de regressão para pequenos sistemas

Jean Carlos Paiva Miranda

JUIZ DE FORA
NOVEMBRO, 2020

Estudo e aplicação de técnicas de teste de regressão para pequenos sistemas

Jean Carlos Paiva Miranda

Universidade Federal de Juiz de Fora

Campus Juiz de Fora

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Heder Soares Bernardino

Co-orientadora: Vânia de Oliveira Neves

JUIZ DE FORA

NOVEMBRO, 2020

Estudo e aplicação de técnicas de teste de regressão para pequenos sistemas

Jean Carlos Paiva Miranda

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO CAMPUS JUIZ DE FORA DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Heder Soares Bernardino
Doutorado em Modelagem Computacional

Vânia de Oliveira Neves
Doutorado em Ciências da Computação e Matemática Computacional

Alessandreia Marta de Oliveira
Doutorado em Computação

Gleiph Ghiotto Lima de Menezes
Doutorado em Computação

JUIZ DE FORA
15 DE NOVEMBRO, 2020

Conteúdo

Lista de Tabelas **8**

Lista de Figuras **9**

Lista de Abreviações **10**

1 Justificativa e Caracterização do Problema **11**
1.1 Caracterização da Pesquisa e Delimitação do Universo 12
1.2 Objetivos 13
1.3 Organização 14

2 Fundamentação Teórica e Revisão Bibliográfica **15**
2.1 Defeitos, erros e falhas 15
2.2 Níveis de Teste 15
2.3 Teste de Regressão 16
2.4 Priorização e Seleção de Casos de Teste 16
2.5 Abordagens para priorização de casos de teste 18
2.5.1 FAST 19
2.5.2 STR 20
2.5.3 I-TSD 20
2.5.4 Abordagem baseada em modelo 21
2.5.5 Abordagem baseada em requisitos 21
2.5.6 Rocket 22
2.6 Priorização de testes e métricas 22
2.7 Considerações finais 23

3 Metodologia **24**

4 Objeto de Estudo **28**

5 Estudo de caso de priorização de casos de teste **32**
5.1 Suítes de Teste 32
5.2 Executando testes utilizando *FAST-PW* 33
5.2.1 *FAST-PW* - suíte 1 34
5.2.2 *FAST-PW* - suíte 2 35
5.2.3 *FAST-PW* com parâmetros alterados - suíte 1 35
5.2.4 *FAST-PW* parâmetros alterados - suíte 2 36
5.3 FAST-one 37
5.3.1 *FAST-one* - suíte 1 37
5.3.2 *FAST-one* - suíte 2 37
5.3.3 *FAST-one* parâmetros de similaridade alterados - suíte 1 38
5.3.4 *FAST-one* parâmetros de similaridade alterados - suíte 2 38
5.4 STR 39
5.4.1 STR - suíte 1 39
5.4.2 STR - suíte 2 39

5.5 I-TSD 40
5.5.1 I-TSD - suíte 1 40
5.5.2 I-TSD - suíte 2 41
5.6 Ordem aleatória 41
5.6.1 Ordem aleatória - suíte 1 41
5.6.2 Ordem aleatória - suíte 2 41
5.7 Testes Ordenados na ordem natural 42
5.7.1 Ordem Natural - suíte 1 42
5.7.2 Ordem Natural - suíte 2 43
5.8 Ordem Inversa 43
5.8.1 Ordem Inversa - suíte 1 43
5.8.2 Ordem Inversa - suíte 2 44
5.9 Conclusões do Experimento 44
5.9.1 Conclusões e Observações - Suíte 1 45
5.9.2 Conclusões e Observações - Suíte 2 47
5.9.3 Considerações finais sobre os experimentos 48

6 Conclusões e Trabalhos Futuros **49**

Bibliografia **51**

Resumo

Existe no mercado de *software* uma pressão cada vez maior para entregas constantes, rápidas e com qualidade. Sabe-se que o processo de teste é crucial para a redução de riscos e aumento na qualidade do produto final. Além disso, o teste, ainda que automatizado, leva tempo e, muitas vezes, este é o recurso mais escasso. Priorizar casos de teste em grandes sistemas de *software* e que contém grandes bases de teste deixa de ser uma opção. No entanto, esse pensamento geralmente não é seguido em um cenário de pequenos sistemas. A fim de investigar a viabilidade dessas técnicas ao se trabalhar com projetos de dimensões menores, este trabalho de conclusão de curso visa avaliar e estudar abordagens que ajudam na priorização de casos de teste para que, assim, o tempo gasto durante o teste de regressão seja reduzido, mas sem perder a qualidade do processo de testagem. Neste trabalho de conclusão de curso, propostas para priorização de casos de teste em sistemas de uma forma geral são discutidas, especialmente quanto às vantagens da adoção dessas estratégias quando usadas em pequenas aplicações. Para que isso pudesse ser realizado, além da definição de estratégias, experimentos com um sistema real de pequeno porte foram conduzidos e os resultados obtidos foram avaliados.

Palavras-chave: teste de *software*, teste de regressão, priorização de casos de teste.

Abstract

There is currently on the software development market the pressure for frequent, faster, and quality deliveries. We know that the test process is crucial for the reduction of risks and the increase of the final product quality. Besides that, testing, even backed by automation, takes time, and that is the most scarce resource. Prioritize test cases on big software systems that contain large test suites might not be an option. However, this line of thought is not usually taken into a small system scenario. In order to investigate the viability of such techniques in small systems, this final paper aims to evaluate and to study approaches that help on the prioritization of test cases, so that, the time invested during the regression testing can be reduced, without losing efficiency of the process of testing. In this final paper, we discussed approaches to prioritize test cases in systems in a general manner and a focus was given to the advantages of using such approaches in a small systems scenario. For this, more than defining strategies, we conducted experimentation with a real system and we analyzed the results obtained.

Keywords: software testing, regression testing, test cases prioritization.

Agradecimentos

Agradeço primeiramente aos meus pais e minhas irmãs por todo apoio dado durante todos esses anos de faculdade. Existe muito sacrifício envolvido nisso, e eu sou e sempre serei eternamente grato por tudo.

Agradeço aos meus amigos, e digo que a minha experiência nesse curso não seria nem de perto a mesma se não fosse por vocês.

Ao professor Heder por ter topado me orientar, e ter tido uma paciência gigante comigo. Obrigado de verdade, sou muito grato e sortudo de ter você aqui.

Gostaria de agradecer também a professora Vânia, que sem sombras de dúvida foi uma das pessoas mais importantes que eu conheci no curso. Obrigado Vânia, o mundo precisa de mais incentivadores como você. Obrigado pelos projetos, pelas conversas, pelas aulas e pelas orientações (e pelos passeios). Obrigado por me empurrar sempre que necessário e obrigado não desistir de mim.

"I will float until I learn how to swim".

Jeff Mangum (The King of Carrot Flowers, Pts. Two & Three)

Lista de Tabelas

3.1 Distribuição de falhas encontradas entre os testes da suítes. 26

5.1 FAST-PW - suíte 1. 34

5.2 FAST-PW - suíte 2. 35

5.3 FAST-PW - suíte 1. 36

5.4 FAST-PW parâmetros alterados - suíte 2. 36

5.5 FAST-one - suíte 1. 37

5.6 FAST-one - suíte 2. 38

5.7 FAST-one - parâmetros de similaridade alterados - suíte 1. 38

5.8 FAST-one - parametros de similaridade alterados - suíte 1. 39

5.9 STR - suíte 1. 39

5.10 STR - suíte 2. 40

5.11 I-TSD - suíte 1. 40

5.12 I-TSD - suíte 2. 41

5.13 *Aleatória* - suíte 1. 42

5.14 *Aleatória* - suíte 2. 42

5.15 *Ordem Natural* - suíte 2. 43

5.16 *Ordem Natural* - suíte 1. 43

5.17 *Ordem Inversa* - suíte 1. 44

5.18 *Ordem Inversa* - suíte 2. 44

Lista de Figuras

3.1 Metodologia adotada 24

4.1 Diagrama da aplicação sob estudo 28

5.1 Teste unitário da suíte 1 - exemplo 1 33
5.2 Teste unitário da suíte 1 - exemplo 2 33
5.3 Número de falhas x APFD - suíte 1 45
5.4 Número de falhas x APFD - suíte 1 - zoom 46
5.5 Número de falhas x APFD - suíte 2 47
5.6 Número de falhas x APFD - suíte 2 - zoom 47

Lista de Abreviações

APFD	Média da porcentagem de falhas detectadas
CD	*Continuos deployment*
CI	*Continous Integration*
CT	Caso de teste
DCC	Departamento de Ciência da Computação
E2E	*End to End*
PCT	Priorização de casos de teste
UFJF	Universidade Federal de Juiz de Fora

1 Justificativa e Caracterização do Problema

Realizar testes em aplicações pode se mostrar uma tarefa trabalhosa e com um grande custo associado no que tange o tempo e recursos de pessoal. Até para cenários de escala reduzida, testes podem demorar muito tempo para serem executados (MARIJAN; GOTLIEB; SEN, 2013). Para auxiliar no processo de testagem, testes automatizados são desenvolvidos. Testes automatizados podem ser definidos como *scripts* que servem para verificar funcionalidades do sistema sob teste. Quando o *software* sob teste sofre alterações, realiza-se o teste de regressão. Durante o teste de regressão (DELAMARO; JINO; MALDONADO, 2013), executa-se novamente testes realizados anteriormente e é verificado se estes testes passaram ou não. Quando tem-se testes automatizados em projetos, executá-los ajuda a economizar tempo da equipe, já que parte do sistema que foi atingido por automações de teste e sendo assim o esforço manual necessário para testagem da aplicação é menor. Em cenários onde a quantidade de testes automatizados é massiva, executar todos eles pode demandar muito tempo. Por isso existem técnicas que ajudam a priorizar as automações para que, assim, falhas sejam descobertas mais cedo e o trabalho para corrigi-las aconteça de maneira mais prematura.

Este trabalho se propõe a estudar técnicas de priorização de casos de teste, e avaliá-las em um software *software* de pequeno porte. Na maior parte das vezes quando se vê propostas para o problema de priorização de casos de teste o foco são situações onde o volume de casos de teste é elevado e que frequentemente extrapola a realidade da maioria das aplicações.

Realizar teste de regressão pode ser decisivo para que seja possível encontrar falhas cedo após mudanças e, consequentemente, corrigi-las. Isso é importante para aplicações de qualquer tipo. Só que em um contexto movimentado de indústria, nem sempre é possível tomar todos os cuidados ou nem sempre se tem tempo para executar todos os testes antes de disponibilizar atualizações para o usuário final. Principalmente em situações delicadas como *deploys*, que é quando uma nova versão de um *software* será disponibilizada para clientes em um contexto de aplicativos por exemplo, ou *hotfixes*, que

seriam correções pontuais em defeitos que foram observados em ambiente de produção, (SHAHIN; BABAR; ZHU, 2017). Utilizando técnicas de priorização de casos de teste, é possível otimizar o esforço de teste nessas duas situações mencionadas, por exemplo.

Uma das principais perguntas que este trabalho se preocupa em responder é: Pode-se utilizar técnicas conhecidas e pensadas para grandes sistemas em um ambiente menor e menos complexo e ainda assim obter as vantagens que as propostas proporcionam?

Na maior parte das vezes, a literatura trata desse problema visando quase que unicamente grandes sistemas ou grandes bases de casos de teste. Algoritmos como *FAST* (MIRANDA et al., 2018) falam de priorização de teste na casa das dezenas de milhares de Casos de Teste (CT), por exemplo. Apesar dos estudos focarem em sistemas de grande porte, existem momentos críticos onde uma testagem rápida e mais direcionada pode ajudar a manter as garantias e diminuir o tempo gasto testando (MARIJAN; GOTLIEB; SEN, 2013), isso sem aumentar os riscos da produção do software. Ressalta-se também que pequenas empresas muitas vezes passam por processos de *CI* (*Continuous Integration*), *CD* (*Continuous Deployment*), e nem sempre essas entregas e atualizações do código acontecem em situações calmas e controladas. Acidentes podem ocorrer e em muitas vezes a necessidade de entregar rapidamente o software existe. Executar todos os testes pode demorar, mas pode-se acelerar a obtenção de respostas referente a falhas dos testes ao se utilizar de abordagens de priorização de casos de teste, como as que são citadas na Seção 2.5.

Chegar nas respostas de abordagens é um dos pontos deste trabalho, e será o foco do Capítulo 5, antes disso, é importante entender a caracterização da pesquisa e a delimitação de universo para este projeto de conclusão de curso, estes dois pontos são explorados na Seção 1.1.

1.1 Caracterização da Pesquisa e Delimitação do Universo

Todo o embasamento das questões e viabilidade da pesquisa vem de artigos e muitas vezes uma abordagem é testada em laboratório utilizando-se sistemas ou bases de teste

conhecidas. Neste trabalho a ideia foi fazer experimentos e aplicá-los em um produto de software real.

A delimitação do universo por sua vez se ocorre no fato de estar-se julgando resultados e baseando as análises em um único caso industrial, o que pode ou não refletir a realidade de empresas do mesmo porte. É difícil então afirmar que os resultados seriam semelhantes em outros ambientes de tamanho similar, a maneira e os papéis que um engenheiro de qualidade ou testador desempenham pode variar, e a maneira que testes são escritos também. Entretanto, é possível ter uma ideia de como sistemas do tipo (aplicações *web*) reagem a adoção deste tipo de estratégia e quais ganhos associados podem surgir desta adoção. Após entender as delimitações e características da pesquisa, pode-se agora listar os objetivos deste trabalho.

1.2 Objetivos

O principal objetivo deste trabalho é aplicar e analisar abordagens de priorização de casos de teste conhecidas da literatura em um contexto de *software* de pequeno porte. Como objetivos específicos deste projeto, pode-se citar:

- Revisão da literatura: Pesquisar e conhecer abordagens pensadas para o problema de priorização de casos de teste automatizados.
- Implementação: Verificar a disponibilidade de código das abordagens selecionadas e, caso não estejam disponíveis, realizar implementações necessárias. Separar os casos de teste do estudo de caso e utilizá-los nas propostas escolhidas.
- Experimentação: Conduzir experimentos considerando diversas formas de entrada, algoritmos e suítes de teste.
- Conclusões: analisar os dados produzidos no processo de experimentação e avaliá-los de acordo com os resultados.

1.3 Organização

Este trabalho está organizado da seguinte forma: no Capítulo 2 são feitas descrições breves de conceitos da área de teste de *software* e priorização de casos de teste, explora-se abordagens de Priorização de Casos de Teste (PCT) existentes na literatura, com foco nos algoritmos que serão utilizados nos experimentos deste trabalho. No Capítulo 3 a metodologia adotada no trabalho é detalhada. O Capítulo 4 descreve a aplicação que originou os casos de teste utilizados no experimento, bem como as características tanto desses CTs quanto do ambiente onde eles eram executados. Também é relatado o processo de desenvolvimento da empresa proprietária deste *software*. Realiza-se no Capítulo 5 experimentos e analisa-se os resultados obtidos. Finalmente, no Capítulo 6 as conclusões são feitas e os trabalhos futuros são elencados.

2 Fundamentação Teórica e Revisão Bibliográfica

Para uma melhor compreensão em priorização de casos de teste, é interessante entender também os conceitos básicos relacionados a teste de *software*. Nesse capítulo conceitos fundamentais da área serão elencados e explicados de maneira breve. Além disso, na Seção 2.4 são discutidas as técnicas de teste de regressão e na Seção 2.5 são enumeradas as abordagens de priorização de casos de teste.

2.1 Defeitos, erros e falhas

Um dos conceitos fundamentais para a área de teste de software é a definição de defeito, erro e falha. **Defeitos** englobam um passo ou processo incorreto; **erros** são a manifestação de um defeito e correspondem a diferença entre os valores esperados de acordo com a especificação e os que foram retornados; **falhas** são saídas incorretas e que não correspondem ao esperado de acordo com à especificação (DELAMARO; JINO; MALDONADO, 2013).

2.2 Níveis de Teste

Pode-se dividir a atividade de teste em níveis (DELAMARO; JINO; MALDONADO, 2013). De acordo com a literatura, tem-se: o nível de unidade, integração e sistema. Nos testes de unidade, a unidade mínima do sistema, por exemplo, métodos ou funções, são testadas de maneira independente e isolada. Testes de integração servem para verificar se as unidades que foram testadas de maneira individual se comunicam como esperado.

O objetivo do teste de sistema é testar o sistema como um todo de maneira integrada, verificando se os requisitos especificados foram atendidos. Dentro dos testes de sistema, tem-se os testes *end-to-end* (E2E), que são geralmente feitos em contextos de sistemas *web*, utilizando um navegador. Nesses casos, os testes *E2E*s simulam a forma que um usuário utilizaria a aplícação normalmente.

2.3 Teste de Regressão

Teste de regressão é um tipo de teste feito para verificar que funcionalidades continuam a funcionar após mudanças no *software* sob teste (DELAMARO; JINO; MALDONADO, 2013). Na indústria, onde há sistemas que evoluem constantemente e novas funcionalidades são incluídas de maneira recorrente, é necessário garantir ou ter maior confiabilidade de que o novo código incluído no programa não introduz novos problemas. Isso faz com que os riscos envolvidos a cada evolução do software sejam reduzidos. Existem técnicas que auxiliam o teste de regressão, seja reduzindo o conjunto de testes, selecionando casos de teste dependendo do contexto, ou alterando a ordem de execução das automações de teste. Na Seção 2.4 pode-se conhecer mais sobre essas técnicas.

2.4 Priorização e Seleção de Casos de Teste

Considerando novamente a evolução constante requerida no mercado de *software* e levando em conta também a popularidade e relevância de métodos ágeis, realizar os testes de regressão apresenta-se como um desafio para a equipe desenvolvedora. O caminho mais simples para lidar com essas mudanças e que trará maior confiabilidade de que não foram introduzidos defeitos no *software* é re-testar todo o sistema utilizando todos os testes disponíveis na base do código, além de replicar os testes manuais. O problema é que podem existir situações que forcem uma rápida testagem. Exemplos de momentos que requerem agilidade maior: um erro grave recém encontrado em ambiente de produção (antes que a atualização seja enviada é preciso verificar que essa alteração não prejudica nenhuma outra funcionalidade), uma alteração de última hora que foi necessária porque uma *API* externa alterou sua resposta, entre outros. Para esses casos, executar todos os casos de teste disponíveis pode ser inviável. Sendo assim, é essencial conseguir oferecer segurança de que as novas mudanças não prejudicam o estado do sistema e, ao mesmo tempo, lidar com limitações de tempo (MARIJAN; GOTLIEB; SEN, 2013).

Em (YOO; HARMAN, 2012) são citadas três abordagens para o auxílio da execução dos testes de regressão. Estas abordagens são: a minimização, a seleção, e a priorização de casos de teste. Na **minimização**, o objetivo é procurar e identificar

casos de teste obsoletos ou que possam ser vistos como redundantes dentro da base de testes. A **seleção** de casos de teste vai selecionar um subconjunto de testes a ser executado para testar apenas as partes do código que foram alteradas. Já a **priorização** de casos de teste preocupa-se em buscar uma ordem de execução ideal. Essa ordem de execução pode ajudar fazendo com que, por exemplo, falhas sejam encontradas primeiro e mais rapidamente, levando a uma maior economia tempo.

Por mais que essas três abordagens possam aparentar semelhanças, é possível traçar claras diferenças entre as mesmas. A minimização é costumeiramente chamada de redução da suíte de testes. Isso significa que, neste caso, a eliminação do teste é permanente. A seleção de casos de teste e a PCT podem também gerar subconjuntos temporários, mas geralmente quando se utiliza minimização de casos de teste, a intenção é que de fato ocorra uma redução real do conjunto de testes. Dessa forma, a definição formal para a abordagem de minimização é (YOO; HARMAN, 2012):

Dados uma suíte de casos de teste T e uma sequência de requisitos R, que vai em $\{r_1, r_2, ..., r_n\}$, que deve ser satisfeita para prover um teste do programa visto como adequado. Tem-se também subconjuntos de T, $\{T_1, T_2, ..., T_n\}$, cada um associado com diferentes r_i, tal que um caso de teste t_j pertencente a T_i possa ser usado para alcançar o requisito r_i. O problema então se resume em achar um conjunto representativo T' cujos casos de teste satisfaçam por completo o conjunto de requisitos. A solução é obtida quando todos os requisitos listados, $\{r_1, r_2, ..., r_n\}$, são satisfeitos.

O problema de minimização de uma suíte de testes pode ser encarado também como um problema NP-completo (GAREY; JOHNSON, 1990) do conjunto-capa. Como dito anteriormente, a minimização é uma solução que visa uma alteração permanente na suíte.

A seleção de casos de teste, por outro lado, trata esse problema de uma forma mais flexível. Há também um subconjunto de testes, mas o contexto de mudanças é considerado. O subconjunto selecionado irá levar em conta partes do sistema que foram alteradas recentemente, por exemplo.

A definição formal da seleção de casos de teste é apresentada a seguir(YOO; HARMAN, 2012): Dado o programa P, a versão modificada P' e uma suíte de teste T.

Sendo assim, tem-se que encontrar um subconjunto de T, T', que é capaz de testar P'. A PCT, conforme discutida anteriormente, não seleciona ou altera o conjunto de testes de nenhuma forma. O que acontece aqui é uma alteração na ordem de execução dos testes. É assumido que eventualmente todos os testes irão ser executados.

A priorização pode ser definida da seguinte forma (YOO; HARMAN, 2012): dado uma suíte de testes T, o conjunto de permutações de T, PT, e uma função de PT para números reais, $f : PT \rightarrow \mathbb{R}$. O problema então se resume em encontrar um $T' \in PT$ tal que $(\forall T")(T" \in PT)(T" \neq T')[f(T') \geq f(T")]$.

Este trabalho tem seu foco abordagens de priorização. Na Seção 2.5 são descritas algumas abordagens de priorização de casos de teste, sendo que algumas dessas abordagens foram selecionadas para os experimentos realizados no Capítulo 5.

2.5 Abordagens para priorização de casos de teste

Nesta seção, serão descritos alguns trabalhos estudados para a condução desta monografia. Esses trabalhos abordam o tema priorização de CTs. Entre os trabalhos estudados, poucos consideraram propostas aplicadas em sistemas web, por exemplo. Em outros trabalhos, o escopo ia além do contexto de empresas de pequeno e médio porte. Em (LEDRU et al., 2012), por exemplo, um dos sistemas considerado continha 43.000 casos de teste, o que é distante da realidade do sistema utilizado neste trabalho, conforme pode ser visto no Capítulo 4.

O *survey* (YOO; HARMAN, 2012) apresenta muitas propostas que poderiam se encaixar no problema. Neste trabalho, as propostas baseadas em modelos (KOREL, 2002) e baseadas em requisitos (SRIKANTH; WILLIAMS; OSBORNE, 2005) foram vistas com mais detalhes apesar de não terem sido consideradas nos experimentos. Nas Subseções a seguir tratam de abordagens de priorização de casos de teste, mas somente 2.5.1, 2.5.2 e 2.5.3 são usadas nos experimentos.

2.5.1 FAST

O *FAST* utiliza a representações em *string* de texto de um caso de teste e tenta comparar com outros casos de teste que existem na suíte. Para chegar na resposta em um baixo tempo de execução, o *FAST* utiliza de *shingling*, *minhashing* e *locality-sensitive hashing* (LSH). O *shingling* (MANBER, 1994) é uma técnica que neste contexto ajuda encontrar similaridades entre textos (CHAUHAN; BATRA, 2014).

A técnica visa representar documentos (no caso, código-fonte), como conjuntos. Dado um documento de texto qualquer, um *k-shingle* é toda possível *substring* consecutiva de tamanho k encontrado dentro dele. Por exemplo, na palavra: *"graduation"* utilizando-se *shingles* de tamanho 3, ficaria como: {*"gra", "rad", "adu", "dua", "uat", "ati", "tio", "ion"*}. Documentos similares terão consequentemente diversos *shingles* em comum. Quando *shingles* são gerados, o volume de dados a serem comparados cresce, e estes se tornam maiores que os conjuntos iniciais. O *FAST* utiliza a similaridade de *Jaccard* para dar os valores de distância entre os itens. Entretanto, esse cálculo fica mais custoso ao adicionar a geração de *shingles* no processo. Para contornar a dificuldade de trabalhar com todos esses dados utiliza-se a técnica de *Minhashing* (LESKOVEC; RAJARAMAN; ULLMAN, 2014). Com *Minhashing* pode-se gerar representações compactas que são chamadas de *signatures*. As *signatures* conseguem preservar os valores de *Jaccard* dos conjuntos que elas representam, fazendo com que o número de pares a serem comparados diminua e o tempo executando o algoritmo seja reduzido. Mesmo com as otimizações que o *Minhashing* proporciona, o número de comparações necessárias para chegar na resposta final ainda pode ser demais. Para isso, o *FAST* utiliza também o algoritmo LSH (LESKOVEC; RAJARAMAN; ULLMAN, 2014), que irá reduzir o escopo das comparações apenas para os itens que têm mais chances de serem parecidos, formando o conjunto candidato. Após estes passos, o FAST gerará as distâncias entre os itens e irá adicionar na solução elementos dissimilares entre si. A ideia é cobrir partes diferentes da aplicação o mais cedo possível para assim conseguir encontrar falhas em diferentes lugares do *SUT*.

Existem diferentes variações do algoritmo *FAST*. Para os experimentos serão utilizados o *FAST-PW* e o *FAST-one*. O *FAST-PW* computa as distâncias entre todos

os pares possíveis dentre os presentes no conjunto candidato e seleciona o caso de teste mais distante possível, ou seja, o caso de teste menos parecido com os já presentes no conjunto solução. A outra variação do *FAST* utilizada nos experimentos é o *FAST-one*. Este último seleciona a cada iteração um caso de teste de maneira aleatória a partir de um conjunto de candidatos.

2.5.2 STR

Outro algoritmo é o STR (LEDRU et al., 2012), proposto baseando-se na hipótese de que a diversidade dos testes ajuda na detecção de falhas no sistema testado. Assim, *STR* utiliza de dados de entrada de casos de teste ou da *string* de texto que representa os testes para gerar as informações de distância. Para calcular a distância entre os dados foi utilizado a distância de *Manhattan*. Para a geração de distâncias entre dados de teste no *STR* não existe avaliação semântica o que, de acordo com o estudo pode levar a falsos negativos. *STR* usa uma função de adequação que é baseada na distância dentre cada caso de teste e o conjunto de casos de teste previamente selecionado. A proposta opta por utilizar uma abordagem gulosa que a cada iteração escolhe um caso de teste mais distante destes que já estão no conjunto solução. Quando há diversos elementos que contêm distância igual entre si, qualquer um destes pode ser escolhidos de maneira aleatória. Segundo os autores, diferentes heurísticas foram testadas para o algoritmo e a heurística gulosa foi a que melhor demonstrou resultados no que diz respeito a escalabilidade (LEDRU et al., 2012).

2.5.3 I-TSD

O algoritmo **I-TSD** (FELDT et al., 2016) também utiliza a diferença entre os casos de teste para realizar a priorização. As comparações são feitas par-a-par e nesta abordagem, a semântica ou sintaxe dos casos de teste também não são consideradas. Esta proposta, diferente das descritas nas Subseções 2.5.1 e 2.5.2 não utiliza representações de *strings* para calcular as distâncias entre casos de teste, mas considera-se o *NCD* (*Normalized Compression Distance*) entre eles. *NCD* (COHEN; VITÁNYI, 2015) é uma forma de calcular a similaridade entre dois itens, independente do que esses dois itens signifiquem.

Isso quer dizer que é possível comparar quaisquer dois casos de teste independente da linguagem que foi escrita, ou do tipo de dado que ambos são feitos. O algoritmo então vai utilizar uma função de compressão, e a representação dessa compressão será utilizada para gerar o conjunto de testes priorizados.

2.5.4 Abordagem baseada em modelo

(KOREL, 2002) introduz um modelo para a resolução do problema de priorização e/ou seleção de casos de teste em que, inicialmente, separam e classificam os casos de teste em conjuntos menores considerando sua prioridade. Na proposta, um caso de teste é considerado com prioridade alta se é relevante, ou seja, se ele, por exemplo, exercita algum arquivo alterado, ou tem algum tipo de ligação com recentes incrementos no modelo. Apesar da proposta contar com mais particularidades e detalhes, para esta pesquisa vale pensar em prioridades de casos de testes e o quão necessário um teste pode ser dependendo do contexto de uma alteração no código fonte.

2.5.5 Abordagem baseada em requisitos

(SRIKANTH; WILLIAMS; OSBORNE, 2005) propõem uma forma de realizar a priorização de casos de teste considerando mapeamentos nos requisitos de um sistema e associando esses requisitos aos conjuntos de teste. O trabalho estabelece o conceito de complexidade de implementação razoáveis, isto é, dependendo da configuração da equipe e do produto, não tomaria um esforço muito extenso. Uma equipe de desenvolvimento muitas vezes sabe estabelecer quais trechos foram alterados pelas últimas atualizações e focar os esforços de teste nestes. É comentado também que as prioridades podem levar em conta o quão relevante essa alteração é para os consumidores do sistema e o quão complexo um requisito pode ser.

Apesar dos pontos positivos, é apontado como ponto fraco dessa solução o fato de que muito do que está envolvido pode ser um tanto subjetivo no que tange a separação de setores e áreas de uma aplicação. Avaliar requisitos mais relevantes e complexos pode sim ser uma tarefa subjetiva, mas é preciso se atentar que esse tipo de método é relativamente viável nos dias de hoje. É possível entender o quão complexo um recurso

do sistema é, ou o quão usado e crítico ele é também. Existem diversas ferramentas que permitem compreender e gerar métricas do sistema em desenvolvimento. Além disso, o próprio contato com o consumidor final pode guiar a equipe a entender as necessidades e compreender os esforços de qualidade necessários para cada situação.

2.5.6 Rocket

(MARIJAN; GOTLIEB; SEN, 2013) apresentam uma proposta e relatam um caso industrial em ambiente de regressão contínua. *ROCKET* é o nome da abordagem e ela se propõe ordenar os casos de teste baseando-se no histórico de falhas. Foi realizado a implantação desse software em conjunto com uma aplicação de código fechado e se constatou que o mesmo foi capaz de revelar 30% mais de falhas para 20% da suíte executada quando comparado com técnicas de priorização de CT manuais.

Após conhecer algumas das abordagens e trabalhos relacionados, é possível pensar nas métricas que serão relevantes para os experimentos deste trabalho. A principal, e que é citada na Seção 2.6, é a Média da porcentagem de Detecção de Falhas (APFD).

2.6 Priorização de testes e métricas

Em (ELBAUM; MALISHEVSKY; ROTHERMEL, 2002), os autores apresentam informações relevantes sobre experimentos realizados utilizando técnicas de priorização. São discutidas métricas avaliadoras, o que é de grande valor para este trabalho. A métrica principal considerada nesta pesquisa é a *APFD*. Os valores podem variar entre 0 a 1 e números maiores implicam em melhores resultados.

Dado uma suíte de testes T contendo n casos de teste e F sendo um conjunto de m falhas reveladas por T. Seja TF_i o primeiro caso na ordenação T' que revela a falha i. O APFD para o T' é então dado pela equação:

$$APFD = 1 - (\frac{TF_1+TF_2+...+TF_m}{nm})$$

Outros trabalhos como (Li; Harman; Hierons, 2007) consideram situações onde não se sabe quais testes revelam quais falhas ou se sequer revelam alguma e, para guiar a priorização, utilizam métricas de cobertura. As métricas são: Porcentagem Média da

Cobertura por Blocos *APBC*; Porcentagem Média da Cobertura de Decisões *APDC*; Porcentagem Média da Cobertura por Linha *APSC*;

Essas métricas de cobertura são "acessíveis", pois muitas bibliotecas de teste oferecem a geração das mesmas, tornando a tarefa consegui-las mais fácil. Apesar de dados de cobertura não exatamente garantirem muito na eficácia dos testes, está ainda é uma métrica importante e muito utilizada no universo de testes de *software*.

2.7 Considerações finais

Neste capítulo foram abordados os trabalhos relacionados a este, as propostas de priorização de casos de teste e as métricas relevantes para este contexto. Também foi descrito um embasamento teórico de teste de *software*. O próximo capítulo trata da metodologia adotada na produção deste trabalho. Também são abordadas as particularidades do experimento e decisões adotadas para o desenvolvimento da pesquisa.

3 Metodologia

Este capítulo trata de informações referentes a metologia, são expostas as decisões de pesquisa, algoritmos candidatos e perguntas a serem respondidas pelos experimentos. A pesquisa descrita neste trabalho tem natureza aplicada. O aspecto de aplicação da pesquisa refere-se a utilização de um software real para usar as abordagens estudadas no decorrer do desenvolvimento deste trabalho. A fim de atender os objetivos especificados na Seção 1.2, esta pesquisa foi dividida em 6 etapas, conforme apresentado na Figura 3.

Figura 3.1: Metodologia adotada

Também foram analisadas de maneira quantitativa os resultados obtidos através dos experimentos que serão mostrados no Capítulo 5.

Inicialmente, foi necessário pesquisar na literatura maneiras de reduzir o tempo investido no processo de teste de regressão. Então, após estudar as propostas candidatas, foi necessário filtrá-las e selecionar aquelas que:

- Não são soluções para uma linguagem específica;
- Utilizam representações dos testes em forma de *string* de texto;

- Não demandam muita alteração da base de testes disponível;

Mesmo existindo um sistema alvo, foi considerado importante que as propostas selecionadas funcionassem independente de linguagem para que as ideias fossem possíveis de reprodução em contextos de diferentes aplicações, independentes das tecnologias usadas. Propostas que utilizavam a *string* que compõe os CTs foram preferidas pois, dessa forma, evitaria a utilização de dados de cobertura e informações reais de falhas, que nem sempre estão disponíveis.

Os experimentos foram realizados utilizando os seguintes algoritmos: *FAST-PW*, *FAST-one* (MIRANDA et al., 2018), *STR* (LEDRU et al., 2012), *I-TSD* (FELDT et al., 2016). Além disso, foram considerados o conjunto de testes na ordem em que aparecem no repositório (chamamos de ordem natural dos dados), ordem inversa, e ordem aleatória (randomizando a ordem aparecem no repositório).

Para a realização dos experimentos, foi preciso inicialmente preparar os dados. Os testes da aplicação sob estudo foram escritos em *PHP*[1] e para poder extrair esses testes e organizá-los de uma forma que fosse possível a utilização junto com os algoritmos de priorização.

Durante a preparação dos dados, todos os espaços e quebras de linhas foram removidos dos casos de testes. O código escrito em *PHP*, linguagem de programação utilizada nos testes, não é prejudicado com a remoção destes caracteres. No entanto, se fosse utilizada outra linguagem para os testes, como *python*[2], o tratamento e separação dos dados deveria ser diferente uma vez que *python* utiliza indentações para identificar escopos dentro de funções ou estruturas. Ainda que esta pesquisa não considera questões sintáticas e semânticas das linguagens, vale ressaltar que casos de teste escritos em *python* poderiam ficar empobrecidos em suas representações por causa da forma de tratamento dos dados escolhida.

Além dos procedimentos listados nos parágrafos anteriores, *scripts* auxiliares foram criados, tanto para organizar os candidatos, quanto para semear falhas e gerar visualizações e informações de métricas.

Cada algoritmo foi executado 30 vezes para que fosse coletado também informações

[1] https://www.php.net/
[2] https://www.python.org/

de desvio padrão. As exceções foram o *STR* e o *I-TSD*, pois o tempo que os mesmos levavam para executar cada leitura podia chegar a até 3400 segundos. No caso desses algoritmos, foram feitas cinco execuções para cada faixa de valor.

Durante os experimentos, foram semeados defeitos para que estes fossem descobertos por testes presentes nas suítes utilizadas no experimento. Esses defeitos são fixos por faixas. Determinou-se que uma porcentagem destes testes seriam capazes de revelar as falhas provocadas por esses defeitos. Foram estipuladas faixas de valores (números de falhas e porcentagens de testes que descobrem falhas) para que fosse possível observar como os valores de APFD se comporta com diferentes concentrações de defeitos e testes que as revelam. Cada algoritmo foi executado considerando estas faixas de valores. Por exemplo, para a execução do *FAST-PW* executou-se inicialmente considerando 90 defeitos e 10% de testes capazes de revelar as falhas. As faixas que utilizadas neste projeto são (número de falhas a serem descobertas - porcentagem de testes capas de descobrir falhas): 90-10%; 45-10%; 30-9%; 10-7%; 5-5%; 3-1%.

Com valores semeados, um teste é capaz de revelar de 0 até n (n sendo o número total de falhas) falhas em uma suíte de teste, como esquematizado na Tabela 3.1. Nas colunas tem-se falhas representadas por números e cada linha representa um caso de teste diferente. Na Tabela 3.1, por exemplo, temos 4 dfeitos e 80% dos casos de teste são capazes de encontrar alguma falha, portanto teríamos a faixa 4-80% nesse cenário.

Tabela 3.1: Distribuição de falhas encontradas entre os testes da suítes.

Esquema de falhas encontradas por casos de teste				
Casos de Teste	Falha 1	Falha 2	Falha 3	Falha 4
Caso 1	X			X
Caso 2				
Caso 3		X	X	X
Caso 4	X	X		
Caso 5	X	X	X	X

Através do números de defeitos injetados e destes casos de teste que revelam falhas, é possível analisar os resultados de *APFD* e avaliar o quão eficiente são os algoritmos para o problema de priorização de casos de teste nesse contexto.

Depois de ter obtido todos os dados e gerado as informações necessárias, a última etapa é analisar os resultados, a partir deles, gerar as conclusões apropriadas. Para os

experimentos realizados, serão consideradas as seguintes métricas:

- **Tempo de execução:** Observa-se o tempo que o algoritmo leva para executar as iterações, com a base de testes;
- **APFD**: essa métrica indica o percentual de falhas encontradas através da vida útil da suíte de teste. Ela é importante para conseguir detectar e entender o quão eficiente uma saída do algoritmo consegue ser em comparações com as outras.

Será observado em quais faixas de valores existem índices mais positivos, como os algoritmos se comportam quando existe uma escassez de testes e falhas e como as respostas variam entre as iterações das execuções das propostas. A partir daí, é possível responder a questão principal que é: qual dos algoritmos consegue priorizar melhor os casos de teste baseando-se em valores de APFD?

A pergunta realizada no parágrafo anterior será respondida utilizando os experimentos aplicados no sistema descrito no Capítulo 4. Além disso, no Capítulo 4 também são descritos detalhes da aplicação e experiências na empresa no que tange o processo de teste. Dessa forma, é possível entender melhor as justificativas para a aplicação de técnicas de priorização de casos de teste.

4 Objeto de Estudo

Neste capítulo é abordado o objeto de estudo desta monografia. Foi utilizada uma aplicação comercial que era desenvolvida com o auxílio do autor desta monografia. O nome do *software* e da empresa não são declarados por motivos de confidencialidade.

Como objeto de estudo, o *software* escolhido foi um aplicativo de pequeno porte. Esse *software* é um sistema *web* baseado em serviços que operava em um contexto de *marketing* digital. Ele é constituído de um sistema central, contendo os controladores e o *frontend*. Além disso, esse serviço central utiliza outros serviços, tanto próprios quanto de terceiros, para que sua funcionalidade principal fosse atendida. A Figura 4 apresenta um diagrama que representa de maneira simplificada a organização do sistema sob teste.

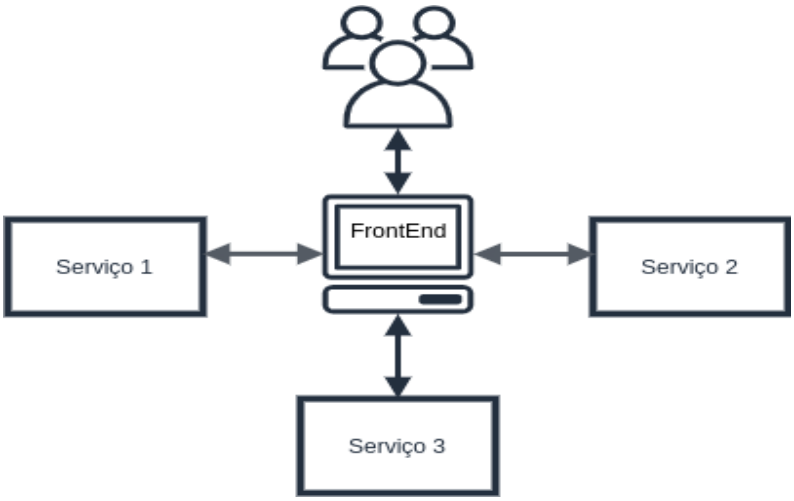

Figura 4.1: Diagrama da aplicação sob estudo

Para realizações de testes automatizados, eram utilizados a biblioteca *PhpUnit*, da linguagem *PHP* e que se assemelha com outras bibliotecas de teste do tipo *xUnit*.

Na empresa, a abordagem com testes e os esforços de automação ocorriam, muitas vezes, em volta do desejo de se obter uma maior cobertura de um serviço, do controlador ou de um determinado módulo. Além disso, existiam testes *E2E* que tentavam fazer verificações de acordo com o que o usuário teria contato. Para os testes *E2E*, utilizava-se *javascript*, e a biblioteca *JEST*[3], além do *puppeteer*[4] (para controle do navegador).

[3]https://jestjs.io/
[4]https://github.com/puppeteer/puppeteer

Além disso, para rodar os testes era necessário a utilização de uma alternativa que nos deixasse executar uma *pipeline* de *CI*, a ferramenta escolhida foi o *Github actions*[5]. Essa ferramenta possibilitava que a equipe criasse um ambiente novo a cada envio de código para o controlador de versão e, assim, era possível ter uma noção do estado atual da aplicação.

Existe uma grande vantagem em se trabalhar assim: poder testar a cada envio e a cada proposição de integração de novo código a base principal é muito cômodo.

Nesta empresa, para que fosse possível obter os resultados dos testes que foram executados na *pipeline* era necessário que todo o processo de inicialização do ambiente fosse feito, juntamente com os serviços de apoio. No geral, o processo automático realizado era o seguinte:

1. Atribuição de variáveis de ambiente e inserção de chaves necessárias nos próximos passos;
2. *Clone* da versão atual no ramo correto do programa principal;
3. *Clone* de todos os outros serviços cada um em seu devido ramo desejado;
4. *Download* de pacotes e dependências necessárias para a execução dos programas;
5. Construção das imagens de *container* para cada serviço desejado (contava também com o *download* de dependências e pacotes necessários para cada repositório);
6. Execução de testes de unidade e integração no *container* da aplicação principal;
7. Geração de relatórios de cobertura e artefatos de teste pós execução;

Todos esses passos, porém, por mais que não numerosos, tomavam uma quantidade de tempo significativa. É importante ressaltar que muitos desses itens demoravam minutos para serem executados e o processo todo podia levar entre 15 a 30 minutos. Apesar de uma primeira impressão parecer pouco, é um longo tempo a se esperar (SHAHIN; BABAR; ZHU, 2017). Vale a pena ressaltar também que isso diminui a velocidade do

[5]https://github.com/features/actions

desenvolvimento de software. Um ciclo parecido com esse também seria executado em um momento de implantação, então ter que esperar duas vezes é ainda mais custoso.

Existem soluções conhecidas para reduzir o tempo que essa *pipeline* demora para ser executada, muitas delas envolvendo *cache* de imagens de *docker*[6], ou pacotes e dependências, por exemplo. Entretanto, um dos maiores gargalos era a execução de testes. Na maioria das vezes, esses testes executavam em tempos bem similares, considerando que não havia defeitos que pudesse causar latências inesperadas. Paraleliza-los poderia ser uma opção, no entanto, a configuração não é trivial. Além disso, muitas das vezes a paralelização pode adicionar complexidade ou problemas nos resultados dos testes. Outro fator é que é recomendado que cada teste deve ser independente visto que uma interferência é possível, caso o desenvolvimento dos CTs não tenha sido feita de maneira correta. Além disso, é possível que testes não tenham sido completamente isolados ou testes de integração tenham um tempo de execução maior dado sua natureza. É possível também que testes mais complexos demorem mais a serem executados, tornando ainda mais difícil otimizar o tempo da fase de testes.

A situação poderia se agravar se nessa *pipeline* fosse adicionada, além dos testes de unidade e integração, testes *E2E*. Como a aplicação acessa serviços externos, a execução desses testes corriqueiramente apresentava comportamentos *flaky*, ou seja, comportamentos não determinísticos em que algumas vezes o caso de teste passa, mas em outras ele sinaliza falha, mesmo que não há defeitos no código (ECK et al., 2019). Os testes E2E também demoram mais (BAI et al., 2001) do que os outros em média porque, além de executar fluxos maiores, é necessário o uso de um navegador *web* (mesmo sem a interface gráfica), o que faz com que mais recursos sejam gastos e mais tempo empregado na tarefa também. A natureza não determinística pode adicionar ainda mais tempo na execução dos testes desse tipo justamente por existir em determinados momentos a necessidade de uma re-teste para confirmar uma falha.

Sendo assim, pode-se observar que a atividade de teste no fluxo de desenvolvimento de software moderno com utilização de *pipelines CI* e testes automatizados pode custar um tempo precioso, ainda que em ambientes de pequeno porte. Todo o processo

[6]http://docker.com/

pode ser ainda mais custoso se falhas forem encontradas em momentos tardios durante a execução de testes. Não é ideal parar a execução dos testes ao detectar a primeira falha, mas já é possível agir diante da mesma. Se essas falhas fossem detectadas de maneira mais rápida, possibilitaria mover recursos humanos a direção da solução do problema encontrado, seja através da remoção do defeito no código, depurações ou até mesmo apenas a comunicação do erro para outros membros do time.

As situações relatadas acima eram situações normais de desenvolvimento, mas há também momentos onde o tempo está muito mais escasso e isso acontecia com frequência na empresa. Em muitas ocasiões, poderia acontecer uma falha crítica no software que já estava disponível para o cliente, ou ainda, um serviço tornar-se indisponível por algum motivo atípico. Nessas situações, era necessário ajustar e enviar um novo código. Esse envio utilizava o mesmo fluxo normal de trabalho, só que dessa vez existe um agravante que é a limitação de tempo e ter que esperar por todos os testes pode se tornar algo prejudicial. Ao mesmo tempo, não rodar os testes pode aumentar os riscos envolvidos. Nesse sentido, a priorização e seleção de casos de teste podem servir de ajuda.

Ao entender como o software sob experimento operava é possível observar melhor onde a priorização de casos de teste poderia vir a agregar. O próximo passo deste trabalho é realizar os experimentos e analisar os valores gerados pelas propostas em cada faixa de valores.

5 Estudo de caso de priorização de casos de teste

Este capítulo tem como objetivo comparar e avaliar os algoritmos de priorização estudados e verificar qual destes se encaixam melhor nas necessidades do estudo de caso apresentado no Capítulo 4. Todos os experimentos foram executados em um AMD Ryzen 5 3600X Cache 32MB 3.8GHz, 16GB de RAM e Sistema Operacional *Linux Mint 20.04*.

5.1 Suítes de Teste

Existem duas suítes de teste para estes experimentos. A primeira suíte foi feita para cobrir uma parte específica do programa original, e os testes que a compõem seguem um padrão que os torna bem similares entre si. Essa semelhança pode ser vista nas Figuras 5.1 e 5.2. A segunda conta com casos de testes mais complexos e variados no que diz respeito a representação das *strings* e não tem uma forma definida como os casos de teste da primeira suíte. O tamanho de cada suíte é como informado a seguir:

- **Suíte 1:** 471 testes
- **Suíte 2:** 445 testes

Para esses experimentos, foram utilizados casos de teste reais que estavam disponíveis para a testagem do aplicativo descrito no Capítulo 4.

```php
public function testGetMetricData_fbads_cpm() {

    $reportsController = new ReportsController();
    $repoCo = new ReflectionClass($reportsController);
    $request = new Request();
    $data['metric_type'] = 6002;
    $data['report_id'] = $this->reports->id;
    $data['security_code'] = $this->reports->security_code;
    $data['source_id'] = $this->integrations->source_id;

    $request->replace($data);
    $r = $reportsController->getMetricData($request);

    $this->assertTrue(isset($r['description']));
}
```

Figura 5.1: Teste unitário da suíte 1 - exemplo 1

```php
public function testGetMetricData_fbads_clicks() {

    $reportsController = new ReportsController();
    $repoCo = new ReflectionClass($reportsController);
    $request = new Request();
    $data['metric_type'] = 6003;
    $data['report_id'] = $this->reports->id;
    $data['security_code'] = $this->reports->security_code;
    $data['source_id'] = $this->integrations->source_id;

    $request->replace($data);
    $r = $reportsController->getMetricData($request);

    $this->assertTrue(isset($r['description']));

}
```

Figura 5.2: Teste unitário da suíte 1 - exemplo 2

5.2 Executando testes utilizando *FAST-PW*

Com as configurações padrões do algoritmo, o resultado é muito similar para todas as execuções feitas. De acordo com o que foi descrito no Capítulo 3, o algoritmo foi executado 30 vezes e em todas as execuções o resultado, no que diz respeito a ordem dos casos de teste na resposta, foi apenas em um teste inicial aleatório seguido por uma lista ordenada dos testes na mesma sequência que eles aparecem no arquivo de entrada. Esse teste inicial aleatório é gerado por padrão pelo algoritmo e utilizado para definir a ordem dos próximos casos de testes. Esse resultado se deu pelo fato de que os testes analisados são semelhantes

entre si em 10% ou mais (de acordo com o LSH). Sendo assim, a forma que o algoritmo dá a resposta acaba por ficar padronizada. A distância entre os testes é igual e o código escolhe por aquele que apareceu primeiro. As subseções a seguir descrevem os resultados para cada suíte de teste com mais detalhes.

É possível também alterar alguns parâmetros nos algoritmos da família *FAST* e forçá-los a fazer uma classificação mais rigorosa. Pode-se alterar a porcentagem da similaridade analisada. Primeiramente utilizou-se um percentual padrão do algoritmo que é de 10%. Após feitas as leituras com estes percentuais, os experimentos foram realizados considerando 72% de similaridade.

5.2.1 *FAST-PW* - suíte 1

Nas 30 execuções para a primeira suíte, foram obtidos os valores APFD médio de 0.9288 e desvio padrão de 0.0081. Além disso, a execução acontece com um tempo médio de 1.6453 segundos, com desvio padrão de 0.03432, como visto na Tabela 5.1. Foram feitas coletas de dados para os seguintes valores: 30 defeitos, e 9% de testes descobridores de falhas (30-9%); 10-7%; 5-5%; 3-1%. Esses parâmetros de números de falhas estão presentes em todas as execuções de algoritmos deste capítulo, a tabela 5.1 apresenta esses dados. Com 45 falhas existentes e 10% dos testes revelando falhas, o APFD passa para 0.9792 em seu valor médio e 0.0052 de desvio padrão. Apensas considerando a alteração no número de falhas, houve uma melhora de cerca de 0.04 no APFD da primeira para a segunda leitura.

FAST-PW com parâmetros iniciais - Suíte 1				
Número de falhas	% de testes falhos	APFD	Desv. Padrão	Tempo de Execução
90	10%	0.9288	0.0081	1.6453
45	10%	0.9792	0.0052	1.7453
30	9%	0.9947	0.0008	1.5341
10	7%	0.9868	0.0037	1.5414
5	5%	0.8922	0.0184	1.6890
3	1%	0.5599	0.0010	1.5860

Tabela 5.1: FAST-PW - suíte 1.

5.2.2 *FAST-PW* - suíte 2

Ao executar a suíte 2 com 90 defeitos e com 10% dos testes tendo falhas obteve-se 0.9844 para a média de APFD e 0.0045 de desvio padrão. Para a segunda suíte, obteve-se 0.9633 e 0.0084, respectivamente para APFD e desvio padrão. Pode-se observar uma queda de APFD quando vamos de 90-10% para 45-10%, isso é a direção contrária do que aconteceu na suíte 1. Essa queda pode ser atribuída ao fato de que o número de falhas caiu, fazendo com que exista uma dificuldade maior do algoritmo em encontrar falhas mais cedo. O restante dos dados da execução de *FAST-PW* para a suíte 2 pode ser visualizado na Tabela 5.2.

FAST-PW com parâmetros iniciais - Suíte 2				
Número de falhas	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9843	0.0045	2.2421
45	10%	0.9633	0.0084	2.2538
30	9%	0.8587	0.0254	2.2936
10	7%	0.9263	0.0152	2.2119
5	5%	0.9211	0.0186	2.2562
3	1%	0.8857	0.0313	2.2863

Tabela 5.2: FAST-PW - suíte 2.

5.2.3 *FAST-PW* com parâmetros alterados - suíte 1

Para 90 defeitos e 10% de testes que são capazes de revelá-las, , obteve-se os valores de 0.9516 para APFD e 0.0019 de desvio padrão. O tempo de execução também se altera chegando a 5.6026 segundos em média, com desvio de 0.0283. O que indicou uma melhora nos valores observados, mesmo com um aumento no tempo de execução, houve uma melhora no APFD, conforme pode ser observado na Tabela 5.3.

Continuando o processo de re-avaliação dos dados considerando parâmetros diferentes, também foram considerados a existência de 45 defeitos com a mesma porcentagem de testes que encontram falhas. Com esses valores, obteve-se uma redução significativa na métrica APFD, apresentando valor médio de 0.9236. O desvio padrão para essa mesma métrica aumentou para 0.00717. Ao utilizar 10 defeitos com a mesma porcentagem de testes que relatam falhas, obtve-se 0.9811 como valor médio de APFD, 0.0058 de desvio.

Nesse caso, é possível notar que o valor de desvio padrão está bem mais alto do que o que foi obtido anteriormente. Apesar disso tem-se um APFD mais alto.

Apesar de ser difícil entender uma regra de como os dados estão se comportando através dessas diferentes medidas e configurações, é possível observar uma mudança na forma em que os testes estão agora dispostos. Se antes existia na saída uma ordem bem semelhante à ordem de entrada, isso já não acontece mais e observa-se diferenças a cada execução do algoritmo.

FAST-PW com parâmetros alterados - Suíte 1				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9516	0.0019	5.6422
45	10%	0.9236	0.0071	5.6761
30	9%	0.9946	0.0006	5.4212
10	7%	0.9844	0.006	5.7468
5	5%	0.9429	0.0105	5.8129
3	1%	0.3398	0.0486	5.4311

Tabela 5.3: FAST-PW - suíte 1.

5.2.4 *FAST-PW* parâmetros alterados - suíte 2

Na Tabela 5.4 pode-se observar como os dados da suíte responderam as alterações dos parâmetros de similaridade. Com exceção dos dados lidos para 30-9% e 10-7%, todas as outras faixas de valores de defeitos e porcentagem de testes que revelam falhas apresentaram valores abaixo do que foram lidos na tabela com parâmetros padrão. Nesse caso, além de valores menores também tem-se um salto no tempo exigido para a geração da resposta, chegando a valores que ficam entre 8,4 e 8,8 segundos.

FAST-PW com parâmetros alterados - Suíte 2				
Número de falhas	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9713	0.0127	11.4676
45	10%	0.9355	0.0206	11.7222
30	9%	0.9451	0.0132	11.6951
10	7%	0.9601	0.0182	11.1066
5	5%	0.9334	0.02984	11.8503
3	1%	0.8455	0.0291	11.5280

Tabela 5.4: FAST-PW parâmetros alterados - suíte 2.

5.3 FAST-one

No *FAST-PW*, as comparações são feitas par-a-par e, a cada iteração, um teste novo é escolhido primeiro e de maneira aleatória.

O *FAST-one* é uma versão alterada do algoritmo *FAST*. A ideia é principalmente observar como ele se comportará ao utilizar os parâmetros iniciais, lembrando que quando o *FAST-PW* foi utilizado, existia uma dificuldade do algoritmo em discernir os elementos, fazendo com que ele simplesmente adicionasse os itens na solução na ordem que os mesmos apareciam.

5.3.1 *FAST-one* - suíte 1

Para a suíte 1, e ao comparar com o *FAST-PW* tem-se em 90-10%, 5-5% e 3-1% valores mais altos de APFD e principalmente para a última existe uma diferença de aproximadamente 0,18. Para 3-1% temos um desvio padrão de 0.1200 bem acima do que foi registrado para o *FAST-PW* para a mesma faixa. A Tabela 5.5 contém mais detalhes sobre os dados.

FAST-one com parâmetros iniciais - Suíte 1				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9573	0.0261	1.1829
45	10%	0.9622	0.0367	1.1118
30	9%	0.9622	0.0260	1.8189
10	7%	0.9543	0.0319	1.2768
5	5%	0.9347	0.0498	1.3499
3	1%	0.7406	0.1200	1.4032

Tabela 5.5: FAST-one - suíte 1.

5.3.2 *FAST-one* - suíte 2

Nesse caso, *FAST-one* se desempenhou melhor que o *FAST-PW* para as faixas de 30-9% e 10-7%. O restante dos dados pode ser visto na Tabela 5.6

FAST-one com parâmetros iniciais - Suíte 2				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9754	0.0195	2.2326
45	10%	0.9440	0.0288	2.419
30	9%	0.9392	0.0381	2.1200
10	7%	0.9355	0.0347	2.4912
5	5%	0.8945	0.0385	2.1019
3	1%	0.8678	0.0872	2.3012

Tabela 5.6: FAST-one - suíte 2.

5.3.3 *FAST-one* parâmetros de similaridade alterados - suíte 1

Novamente aqui tem-se valores acima de 0.90 para APFD, com exceção da faixa 3-1%. Nessa faixa obteve-se 0.7133 o que é menos do que foi obtido usando o *FAST-one* com os parâmetros padrão. Entretanto, é possível ver que tem-se para essa faixa com o *FAST-one* valores melhores do que foram vistos quando *FAST-PW* foi utilizado, tanto para parâmetros de similaridade originais quanto para os alterados.

FAST-one com parâmetros alterados - Suíte 1				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9496	0.0243	5.4147
45	10%	0.9553	0.0306	5.5394
30	9%	0.9651	0.0155	5.299
10	7%	0.9513	0.0394	5.716
5	5%	0.9314	0.0466	5.1841
3	1%	0.7133	0.1623	5.9301

Tabela 5.7: FAST-one - parâmetros de similaridade alterados - suíte 1.

5.3.4 *FAST-one* parâmetros de similaridade alterados - suíte 2

Com exceção da primeira e da última faixa, *FAST-one* com parâmetros de similaridade alterados tem desempenho melhor no que diz respeito ao APFD do que o *FAST-one* original, conforme pode ser observado na Tabela 5.8.

FAST-one com parâmetros alterados - Suíte 2				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9561	0.0413	11.8932
45	10%	0.9506	0.0356	11.1280
30	9%	0.9529	0.0343	11.1208
10	7%	0.9567	0.0303	11.4320
5	5%	0.9340	0.0558	11.2130
3	1%	0.7752	0.1465	11.5501

Tabela 5.8: FAST-one - parametros de similaridade alterados - suíte 1.

5.4 STR

Nas subseções a seguir são analisados os dados gerados ao se executar os testes utilizando o algoritmo *STR*.

5.4.1 STR - suíte 1

Para primeira suíte, obteve-se resultados entre 0.8 e 0.96 para todas as faixas listadas. Para a faixa 3-1%, os resultados obtidos foram melhores do que os vistos para o *FAST-one*, tanto para os parâmetros originais, quanto para quando a execução foi feita com parâmetros alterados, *STR* na faixa 3-1%, também foi superior ao *FAST-PW* tanto para parâmetros originais quanto para os parâmetros alterados. Esses dados podem ser visualizados na Tabela 5.9.

STR - Suíte 1				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9467	0.0350	10.9448
45	10%	0.9519	0.0143	10.9280
30	9%	0.9633	0.0152	10.9174
10	7%	0.9640	0.0305	10.9206
5	5%	0.9288	0.0389	10.9225
3	1%	0.8004	0.1011	10.9224

Tabela 5.9: STR - suíte 1.

5.4.2 STR - suíte 2

Ao executar os testes com o *STR* na suíte 2, foram obtidos resultados que também são superiores a 0.9 de APFD, com exceção da faixa 3-1%. Nesta faixa obtivemos 0.6708, que

é (para faixa) o menor resultado para qualquer algoritmo usando a suíte 2. Além disso, é possível observar um salto no tempo de execução, chegando a 983 segundos para qualquer faixa da suíte 2, como pode ser visto na Tabela 5.10.

STR - suíte 2				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9351	0.02907	983.3816
45	10%	0.9866	0.0013	983.3984
30	9%	0.9541	0.01183	983.4000
10	7%	0.9643	0.0100	983.4403
5	5%	0.9288	0.0329	983.3511
3	1%	0.6708	0.1471	983.8547

Tabela 5.10: STR - suíte 2.

5.5 I-TSD

Foi feito também a execução do algoritmo I-TSD, conforme apresentado nas subseções a seguir. Nesse algoritmo é possível notar valores semelhantes no que diz respeito a APFD, só que com a diferença nos valores de tempo de execução.

5.5.1 I-TSD - suíte 1

Obteve-se na suíte 1 resultados que variam entre 0.8 e 0.98, a faixa 3-1% teve 0.7467 (Tabela 5.11) de APFD o que significa que é a segunda melhor para a faixa (em termos de APFD). Entretanto, os valores para tempo de execução do algoritmo aqui sobem de maneira significante se comparado a outras abordagens aplicadas na suíte 1.

I-TSD - suíte 1				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9286	0.0360	145.2051
45	10%	0.9525	0.0271	140.9839
30	9%	0.9558	0.02789	136.1818
10	7%	0.9739	0.0154	133.9032
5	5%	0.9328	0.0250	140.0422
3	1%	0.7467	0.0848	137.3867

Tabela 5.11: I-TSD - suíte 1.

5.5.2 I-TSD - suíte 2

Na execução do algoritmo *I-TSD* para a segunda suíte foi possível observar resultados semelhantes a outras abordagens para a suíte 2. No entanto, aqui os resultados foram ruins ao se considerar o tempo de execução do algoritmo. Cada resultado demorou cerca de 3100 segundos para retornar resposta, fator que por si só, impossibilita a utilização desta abordagem para essa suíte. Os dados dessa execução podem ser vistos na Tabela 5.12.

I-TSD - suíte 2				
Número de defeitos	% de testes falhos	APFD	Desv. Padrão	Tempo de execução
90	10%	0.9265	0.0196	3096.4033
45	10%	0.9600	0.0300	3096.4033
30	9%	0.9649	0.0313	3124.3228
10	7%	0.9292	0.0441	3131.5371
5	5%	0.9166	0.0358	3211.8823
3	1%	0.6890	0.1471	3114.9138

Tabela 5.12: I-TSD - suíte 2.

5.6 Ordem aleatória

Ao executar os testes utilizando a ordem aleatória, observou-se que os valores não se diferem tanto se comparado com os que foram lidos anteriormente.

5.6.1 Ordem aleatória - suíte 1

Para a suíte 1, é possível ver, na faixa 3-1%, valores de APFD acima de 0.70. Esse valor não foi alcançado pelo *FAST-PW* em nenhuma das ocasiões em que foi utilizado para a suíte 1. Os dados obtidos para as demais faixas de valores podem ser vistos na Tabela 5.13.

5.6.2 Ordem aleatória - suíte 2

Os valores lidos para a suíte 2 se assemelham com os lidos nas propostas da literatura. No entanto, aqui não houve destaque nas leituras de APFD: com exceção da faixa a 3-1%, as demais obtiveram os valores entre 0.9-1, conforme pode ser visto na Tabela 5.14.

Aleatória - Suíte 1			
Número de defeitos	% de testes falhos	APFD	Desv. Padrão
90	10%	0.9609	0.0196
45	10%	0.9556	0.0256
30	9%	0.9477	0.0371
10	7%	0.9587	0.0305
5	5%	0.9208	0.0562
3	1%	0.7297	0.1279

Tabela 5.13: *Aleatória* - suíte 1.

Aleatória - Suíte 2			
Número de defeitos	% de testes falhos	APFD	Desv. Padrão
90	10%	0.9431	0.0402
45	10%	0.9570	0.0179
30	9%	0.9587	0.0316
10	7%	0.9527	0.0228
5	5%	0.9253	0.0630
3	1%	0.7352	0.1794

Tabela 5.14: *Aleatória* - suíte 2.

5.7 Testes Ordenados na ordem natural

Nesta seção são feitas as análises considerando a execução dos testes na ordem de apresentação (ordem natural), tanto a original quanto na inversa.

5.7.1 Ordem Natural - suíte 1

Ao executar as suíte, foi possível observar novamente o quão similar esses dados são. Para a suíte 1 na ordem natural, obteve-se valores de APFD acima de 0.95. Isso muda quando a análise é realizada considerando os dados na faixa 5-5%. A partir dessa faixa, os valores caem para menos de 0.9 e permanecem assim até a faixa 3-1%. Quando isso acontece, os valores de APFD vão cair mais do que 0.1 quando comparado com os parâmetros anteriores. Essa queda continua ao considerar 3 falhas e 1% parâmetros, chegando a 0.8418 de APFD, conforme pode ser visto na Tabela 5.15.

Ordem Natural - Suíte 1		
Número de defeitos	Porcentagem de testes que revelam falhas	APFD
90	10%	0.9525
45	10%	0.9719
30	9%	0.9966
10	7%	0.9875
5	5%	0.8894
3	1%	0.8418

Tabela 5.15: *Ordem Natural* - suíte 2.

5.7.2 Ordem Natural - suíte 2

Na suíte 2, obteve-se 0.5150 de APFD com apenas 3 falhas e 1% de CT que revelam falhas, o que nos dá indicações que os casos de testes que encontravam falhas estavam mais ao fim da suíte de teste. Os valores obtidos para essa execução podem ser vistos na Tabela 5.16.

Ordem natural - Suíte 2		
Número de defeitos	Porcentagem de testes que revelam falhas	APFD
90	10%	0.9528
45	10%	0.9737
30	9%	0.9502
10	7%	0.9151
5	5%	0.8912
3	1%	0.5150

Tabela 5.16: *Ordem Natural* - suíte 1.

5.8 Ordem Inversa

5.8.1 Ordem Inversa - suíte 1

Para a ordem inversa, a suíte 1 obteve 0.9944 de APFD (90 falhas e 10% de CT reveladores de falha) o que é um valor que nenhum outro algoritmo ou ordem de execução alcançou. Por outro lado, na faixa de 3 defeitos e 1% de testes acusando essas falhas, obteve-se 0.5899 de APFD, representando a segunda leitura mais baixa dentro todas coletadas nesta faixa. A Tabela 5.17 apresenta os dados dessa execução.

Ordem Inversa - Suíte 1		
Número de defeitos	Porcentagem de testes que revelam falhas	APFD
90	10%	0.9944
45	10%	0.9345
30	9%	0.9895
10	7%	0.9809
5	5%	0.9896
3	1%	0.5899

Tabela 5.17: *Ordem Inversa* - suíte 1.

5.8.2 Ordem Inversa - suíte 2

A suíte 2, quando executada na ordem inversa, também alcançou valores acima de 0.99, para 30-9% e 10-9%. Uma diferença aqui é que não teve uma queda de APFD para abaixo de 0.85. Este valor de APFD aparece justamente ao se executar com os parâmetros 90-10%, o que indica uma concentração maior de testes que revelam falhas no início da suíte. Os resultados dessa execução podem ser observados na Tabela 5.18.

Ordem Inversa - Suíte 2		
Número de falhas	Porcentagem de testes que revelam falhas	APFD
90	10%	0.8551
45	10%	0.9689
30	9%	0.9944
10	7%	0.9930
5	5%	0.9315
3	1%	0.9135

Tabela 5.18: *Ordem Inversa* - suíte 2.

5.9 Conclusões do Experimento

É possível observar que os diferentes algoritmos e linhas de base de comparação (ordem aleatória, natural e inversa) mostraram valores próximos a 1 de APFD em momentos diferentes. Por exemplo, para a suíte 1, utilizando o algoritmo *FAST-PW* com parâmetros originais, obteve-se, para a faixa 30-9%, um APFD de 0.9947, o que é muito próximo de 1. Esse foi um dos maiores valores lidos, independente da faixa, para todas as leituras do experimento. No entanto, esse mesmo algoritmo, com as mesmas configurações e na mesma suíte, obteve 0.5599 para a faixa 3-1%, o que é um dos menores valores da faixa

para qualquer leitura feita neste experimento.

5.9.1 Conclusões e Observações - Suíte 1

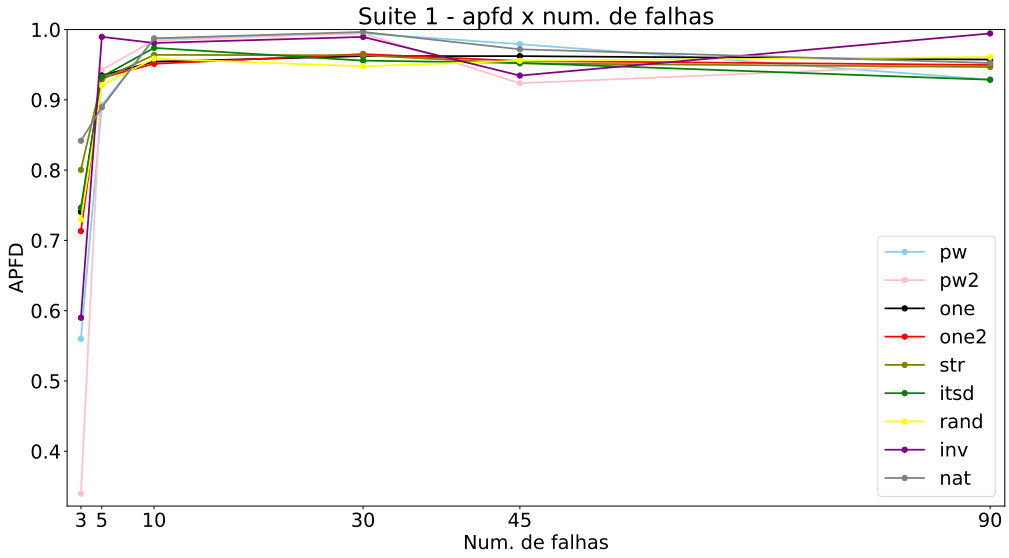

Figura 5.3: Número de falhas x APFD - suíte 1

Para a suíte 1, através das faixas de valores de falhas e testes que revelam falhas, o melhor valor de APFD, em média, considerando todas as faixas foi 0.94. Este valor foi obtido quando se utilizou a ordem natural dos casos de teste, ou seja, executar os testes da forma em que os mesmos eram geralmente dispostos. Sendo assim, mostrou-se a melhor alternativa para a suíte 1. É possível notar que para as faixas acima de 3-1% e 5-5% os valores se estabilizam entre 1 e 0.8, é possível visualizar esses dados nas Figura 5.3 e 5.4. Além disso, há um valor mínimo para o FAST-PW com parâmetros alterados, que é 0.33. Este candidato também apresenta o menor valor na faixa 45-10%.

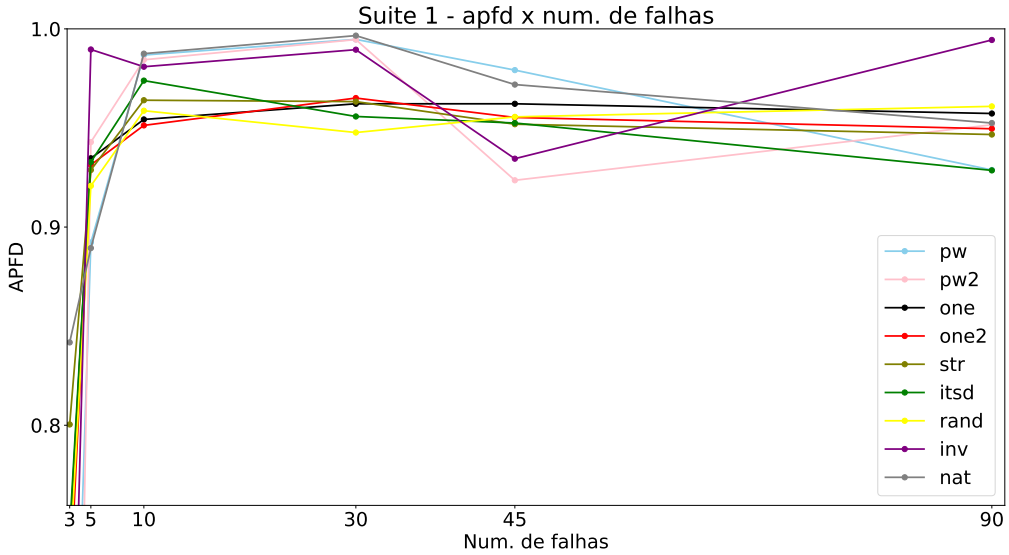

Figura 5.4: Número de falhas x APFD - suíte 1 - zoom

A ordem inversa obteve o maior valor para a primeira suíte (quando se considera os valores de APFD), chegando a 0.94. É interessante também observar as médias de APFD para os outros candidatos utilizando a suíte 1. STR obteve 0.9259 de média, *FAST-one*: 0.9186; *I-TSD*: 0.9151; Ordem Inversa: 0.9131; Ordem Aleatória: 0.9122; *FAST-one*-2: 0.9110; *FAST-PW*: 0.8903; *FAST-PW*-2: 0.8562;

Observa-se um desempenho pior para o *FAST-PW* se comparado aos outros algoritmos e considerando a suíte 1. *STR*, *FAST-one* e*I-TSD* lideram em média de APFD (após da ordem natural), mas tanto o *STR* e *I-TSD* tem performance ruim no que diz respeito ao tempo de execução, principalmente se comparado ao *FAST-one*. Por mais que apresentem bons valores de APFD, a adoção dos mesmos se torna muito difícil pelo tempo médio de resposta.

5.9.2 Conclusões e Observações - Suíte 2

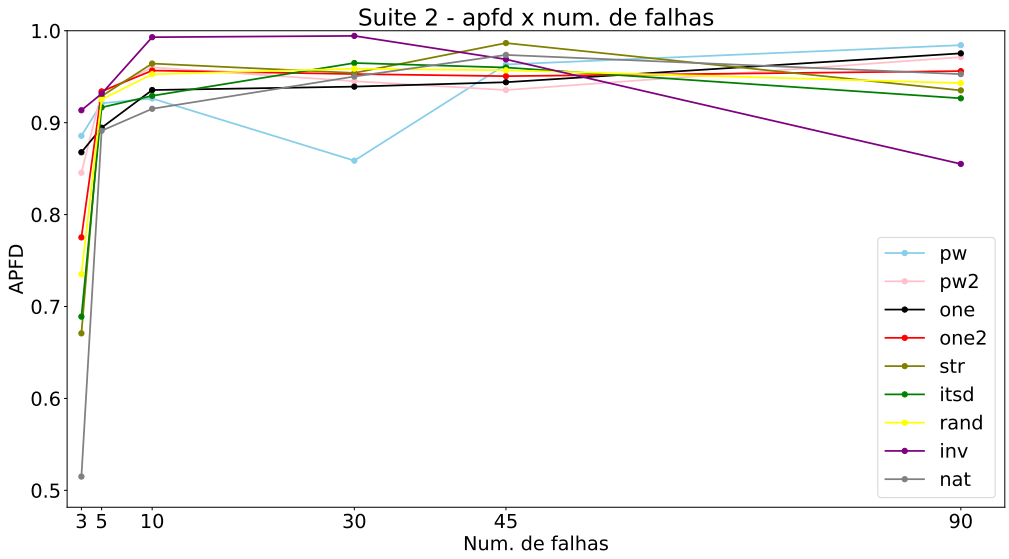

Figura 5.5: Número de falhas x APFD - suíte 2

O menor valor lido entre todas as abordagens foi 0.55 para a execução na ordem natural, também na faixa 3-1% (na suíte 1, esta faixa foi responsável também pelos menores valores de APFD), pode-se visualizar os dados da suíte 2 através dos gráficos nas Figuras 5.5 e 5.6.

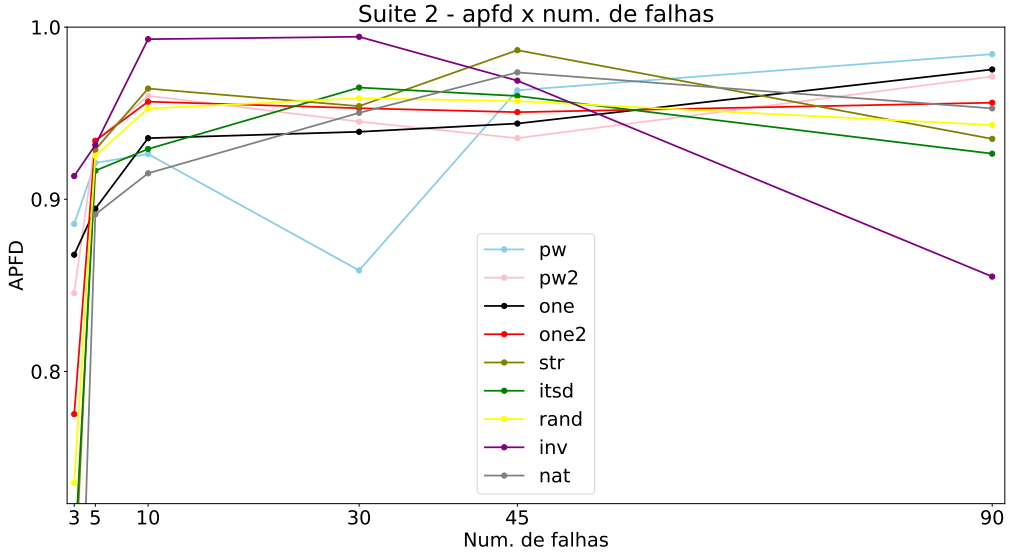

Figura 5.6: Número de falhas x APFD - suíte 2 - zoom

Para a faixa 30-9%, o *FAST-PW* obteve o menor valor, distante dos outros al-

goritmos. Ao utilizar a ordem inversa para a execução dos testes na suíte 2, foi obtido, nas faixas 10-7% e 30-9%, valores acima de 0.9. A execução na ordem inversa também rendeu a maior média de APFD, chegando a 0.943. Pode-se ver as outras médias de APFD em ordem decrescente para a suíte 2 a seguir: *FAST-PW*-2: 0.9318; *FAST-one*: 0.9261; *FAST-PW*: 0.9232; *FAST-one*-2: 0.9209; Ordem Aleatória: 0.9120; *STR*: 0.9066; *I-TSD*: 0.8977; Ordem Natural: 0.8663;

5.9.3 Considerações finais sobre os experimentos

Ao observar as duas suítes, é possível notar que tanto as abordagens trazidas da literatura, quanto as linhas de base conseguem, nas bases de testes disponíveis, retornar resultados de APFD acima de 0.8 para a maioria das faixas de valores analisados. É necessário considerar que, em aplicações reais, o número de testes que revelam falhas e o número de defeitos possivelmente não seriam tão altos. O *FAST* (MIRANDA et al., 2018), *I-TSD* (COHEN; VITÁNYI, 2015) e *STR* (LEDRU et al., 2012), quase sempre tinham desempenho superior aos outros algoritmos a que foram comparados. No entanto, neste trabalho, todos eles em determinados momentos tiveram resultados piores para a ordem natural e para a ordem de execução inversa, para as suítes 1 e 2, respectivamente.. É difícil apontar uma causa única para o resultado observado, mas é interessante notar como a capacidade e a eficácia de uma proposta pode variar dependendo do contexto de aplicação. Neste trabalho, que considerou um contexto de suítes de teste menores, as abordagens da literatura têm dificuldade para desempenhar tão bem quanto as linhas de base usadas. A causa poderia ser o tamanho da suíte ou o formato dos testes, mas seria necessário um estudo mais aprofundado para determinar as verdadeiras razões .

6 Conclusões e Trabalhos Futuros

Neste trabalho discutiu-se propostas de priorização de casos de teste, aplicando-as em um sistema real. Avaliou-se os resultados e observou-se que para as suítes de teste disponíveis e que foram utilizadas no experimento, nenhum dos algoritmos da literatura conseguiu demonstrar clara superioridade no que diz respeito a valores a valores de APFD.

Através dos experimentos realizados foi possível observar que muitas vezes as propostas não conseguiram levar a um lugar claro de como organizar de fato as suítes de teste. Isso em grande parte pode ser atribuído ao fato de que os conjuntos de casos de teste disponíveis não serem grandes o suficiente para que seja possível colher de fato estes benefícios. A ideia deste projeto foi aplicar essas propostas em um software pequeno e entender como elas se comportariam em um contexto deste tipo. Uma dos pontos percebidos nos experimentos realizados neste trabalho foi que na maioria dos cenários, independente da ordem da suíte, existiu pouca variação no APFD observado. Mesmo quando existiam poucos defeitos semeados e poucos casos de teste que descobriam falhas, os resultados ainda eram positivos quando olhados de maneira isolada. Realizar a execução do algoritmo *FAST-one*, por exemplo, em uma escala de precisão maior, poderia levar mais tempo e o ganho não foi tão alto se comparado ao que ele demonstra no contexto em que foi projetado.

A maior parte dos trabalhos de priorização de casos de teste, como dito anteriormente neste trabalho, está focado em grandes projetos, ou repositórios famosos, mas é importante considerar também estratégias que otimizam a fase de testes em *softwares* de menor escala. Este trabalho pode ser considerado como um primeiro passo nesse sentido.

Para o futuro, é importante analisar melhor a organização e outras formas de integrar esses tipos de abordagens no ciclo de desenvolvimento do código. Muito das bibliotecas de teste não oferecem por definição históricos de falhas e informações deste tipo. Uma possibilidade seria replicar o estudo com outras aplicações menores e de contextos diferentes, explorar outros algoritmos, considerar testes de outros tipos.

Aplicar abordagens do tipo das que foram estudadas aqui em outros contextos,

como sistemas mais complexos, como por exemplo, sistemas de sistemas, também é uma proposta para trabalhos futuros. Entender como fazer testes de regressão nesses sistemas ainda é um desafio.

Bibliografia

BAI, X. et al. Distributed end-to-end testing management. In: *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. [S.l.: s.n.], 2001. p. 140–151.

CHAUHAN, S. S.; BATRA, S. Finding similar items using lsh and bloom filter. In: *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies*. [S.l.: s.n.], 2014. p. 1662–1666.

COHEN, A. R.; VITÁNYI, P. M. B. Normalized compression distance of multisets with applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 37, n. 8, p. 1602–1614, 2015.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013.

ECK et al. Understanding flaky tests: The developer's perspective. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 830–840. ISBN 9781450355728. Disponível em: ⟨https://doi.org/10.1145/3338906.3338945⟩.

ELBAUM, S.; MALISHEVSKY, A. G.; ROTHERMEL, G. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, v. 28, n. 2, p. 159–182, 2002.

FELDT, R. et al. Test set diameter: Quantifying the diversity of sets of test cases. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2016. p. 223–233.

GAREY, M. R.; JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN 0716710455.

KOREL, B. Model based regression test reduction using dependence analysis. In: . [S.l.: s.n.], 2002.

LEDRU, Y. et al. Prioritising test cases with string distances. *Autom. Softw. Eng.*, v. 19, p. 65–95, 03 2012.

LESKOVEC, J.; RAJARAMAN, A.; ULLMAN, J. D. *Mining of Massive Datasets*. 2nd. ed. USA: Cambridge University Press, 2014. ISBN 1107077230.

Li, Z.; Harman, M.; Hierons, R. M. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, v. 33, n. 4, p. 225–237, 2007.

MANBER, U. Finding similar files in a large file system. In: *USENIX WINTER 1994 TECHNICAL CONFERENCE*. [S.l.: s.n.], 1994. p. 1–10.

MARIJAN, D.; GOTLIEB, A.; SEN, S. Test case prioritization for continuous regression testing: An industrial case study. In: *2013 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2013. p. 540–543.

MIRANDA, B. et al. Fast approaches to scalable similarity-based test case prioritization. In: *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 222–232. ISBN 978-1-4503-5638-1. Disponível em: ⟨http://doi.acm.org/10.1145/3180155.3180210⟩.

SHAHIN, M.; BABAR, M. A.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, v. 5, p. 3909–3943, 2017.

SRIKANTH, H.; WILLIAMS, L.; OSBORNE, J. System test case prioritization of new and regression test cases. In: . [S.l.: s.n.], 2005.

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 22, n. 2, p. 67–120, mar. 2012. ISSN 0960-0833. Disponível em: ⟨http://dx.doi.org/10.1002/stv.430⟩.