UNIVERSIDADE DO ESTADO DA BAHIA

DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

ANA CECÍLIA SANTOS SOUZA

AI+RTESTING: UM MÉTODO PARA REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE INTEGRANDO AS FERRAMENTAS JABUTI, MUJAVA E WEKA

SALVADOR

2018

ANA CECÍLIA SANTOS SOUZA

AI+RTESTING: UM MÉTODO PARA REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE INTEGRANDO AS FERRAMENTAS JABUTI, MUJAVA E WEKA

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Área de Concentração: Sistemas de Informação
Linha de Pesquisa: Teste de Software

Orientador: Msc. Alexandre Rafael Lenz

SALVADOR

2018

FICHA CATALOGRÁFICA

Sistema de Bibliotecas da UNEB
Dados fornecidos pelo autor

S729a Souza, Ana Cecília Santos

AI+RTesting: Um Método para Remoção de Redundâncias de Conjuntos de Casos de Teste Integrando as Ferramentas Jabuti, Mujava e Weka / Ana Cecília Santos Souza.-- Salvador, 2018.
79 fls.

Orientador(a): Msc. Alexandre Rafael Lenz.
Inclui Referências
TCC (Graduação - Sistemas de Informação) - Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra. Câmpus I. 2018.

1.Sistemas de Informação. 2.Teste de Software. 3.Teste de Regressão. 4.Aprendizado de Máquina. 5.Redução de Conjuntos de Casos de Teste.

CDD: 004

ANA CECÍLIA SANTOS SOUZA

AI+RTESTING: UM MÉTODO PARA REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE INTEGRANDO AS FERRAMENTAS JABUTI, MUJAVA E WEKA

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.
Área de Concentração: Sistemas de Informação
Linha de Pesquisa: Teste de Software

Aprovada em: 28 de Novembro de 2018

BANCA EXAMINADORA

Prof. Msc. Alexandre Rafael Lenz (Orientador)
Universidade do Estado da Bahia - UNEB

Prof. Dr. Mônica de Souza Massa
Universidade do Estado da Bahia - UNEB

Prof. Dr. Eduardo Manuel de Freitas Jorge
Universidade do Estado da Bahia - UNEB

AGRADECIMENTOS

Agradeço, primeiramente, à Deus por ter sido meu mantenedor durante toda a graduação. Apesar de ser comum as pessoas se afastarem da fé quando ingressam no meio acadêmico, no meu caso, foi exatamente o oposto. Para mim, isso é a coisa mais importante. Agradeço aos meus pais por todo o apoio e incentivo que eles me deram e, principalmente, por ter suportado todo minha ausência e extresse nesses últimos anos. Agradeço ao Prof^{o} Alexandre Lenz, meu orientador, por todo o esforço que ele dedicou a este grupo de pesquisa e por toda paciência que ele teve comigo. Agradeço a Prof^{a} Debora Rêgo pelos conselhos, pelo suporte e pelas correções. Agradeço à Thamis por ter sido uma das minhas maiores incentivadoras desde os primórdios. Agradeço aos amigos de graduação por ter estado ao meu lado na alegria e na tristeza. Thamires, você é uma das minhas maiores inspirações. Só Deus pode retribuir tudo o que você fez por mim durante nossos anos de graduação e até mesmo depois. Obrigada mesmo, amiga! Elizabete, você também esteve comigo desde do começo. Obrigada por tudo! Lucas, foi uma surpresa muita grande encontrar um conterrâneo na UNEB. Apesar das nossas personalidades serem muito diferente, você se tornou o meu melhor amigo. Eu te agradeço por toda a companhia, o apoio e a força que você me deu todos esses anos. Augusto, você realmente é a representação da frase: "Amigo é para essas coisas". Sem dúvidas, você deve ter sido uma das pessoas que mais incomodei nessa graduação. Obrigada por toda a paciência, a disponibilidade, o apoio e, principalmente, a sua amizade. Stephanie, eu te agradeço muito todas as horas que passamos juntas fazendo trabalhos que pareciam infinitos e por ter sendo uma grande amiga. Obrigada! Eder, obrigada por toda apoio durante esse TCC. Se não fosse sua ajuda, esse trabalho não tinha terminado. Agradeço, por fim, a todos que direta ou indiretamente contribuíram para a minha formação pessoal e acadêmica.

"Where were you when I laid the foundations of the earth? When the morning stars sang together and all the sons of God shouted for joy?"

(The Book of Job)

RESUMO

A execução do teste de regressão é imprescindível para garantir que adições e alterações realizadas em um software não introduzam erros em funcionalidades já existentes. Consequentemente, o teste de regressão é uma atividade dispendiosa e executada com frequência. Em vista disso, metodologias foram propostas para reduzir o esforço empregado na execução dessa técnica de teste, dentre essas metodologias encontra-se a redução de casos de testes. A redução consiste na remoção permanente de casos de testes redundantes do conjunto casos de testes original. Com intuito de auxiliar na identificação desses casos de teste redundantes, podem ser aplicados algoritmos de aprendizado de máquina. Através dos quais é possível associar as informações obtidas durante a atividade de teste para detectar comportamentos análogos. Assim sendo, o objetivo desse trabalho foi a análise, implementação e validação de um método automatizado para remoção de redundâncias em um conjunto de casos de teste que tem por finalidade a redução do esforço dispendido pelo testador durante a execução do teste de regressão em métodos implementados na linguagem Java. A automação consistiu em coletar as informações da atividade de teste fornecidas pela integração das ferramentas de teste Jabuti e Mujava. Subsequentemente, esse dados foram submetidos aos algoritmos de agrupamento e classificação disponíveis no pacote de software Weka. Com a obtenção das classes de equivalências que foram geradas através das regras de classificação, foi aplicado o cálculo da Progressão Aritmética (PA) para selecionar os casos dos teste que devem ser removidos do conjunto de teste a ser reexecutado. Após a realização dos experimentos de validação, foi possível concluir através dos resultados obtidos que o método automático conseguiu remover as redundâncias atingindo o mesmo valor dos critérios estruturais e escore de mutação do conjunto de casos de teste original.

Palavras-chave: Qualidade. Teste de Software. Teste de Regressão. Aprendizado de Máquina. Redução.

ABSTRACT

Running the regression test is imperative to ensure that additions and changes made to a software have not introduced errors in existing functionality. Consequently, regression testing is a costly and frequently performed activity. Therefore, methodologies were proposed to reduce the effort employed in the execution of this test technique, the reduction of test cases is one of these methodologies. The reduction comprises in the permanent removal of redundant test cases from the original test cases set. In order to assist in the identification of these redundant test cases, machine learning algorithms can be applied through which it is possible to associate the information obtained during the test activity to detect similar behaviors. Thus, the objective of this work was the analysis, implementation and validation of an automated method to remove redundancies in a set of test cases that has the purpose of reducing the effort expended by the tester during the execution of the regression test in implemented methods in the Java language. The automation consisted of collecting the test activity information provided by the integration of the Jabuti and Mujava test tools. Subsequently, this data was submitted to the clustering and classification algorithms available in the software Weka. By obtaining the classes of equivalences that were generated through the classification rules, the Arithmetic Progression (AP) calculation was applied to select the test cases that should be removed from the test set to be reexecutted. After the validation experiments, it was possible to conclude from the obtained results that the automatic method was able to remove the redundancies reaching the same value of the structural criteria and mutation score of the original set of test cases.

Keywords: Quality. Software Testing. Regression Testing. Machine Learning. Reduction.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Atualmente, evidencia-se a indissociabilidade dos *softwares* no cotidiano das pessoas. Em virtude disto, as exigências por *softwares* com maior qualidade, que é a capacidade do produto de *software* em satisfazer as necessidades implícitas e explícitas quando usado em condições específicas (ISO/IEC, 2010), têm motivado vários pesquisadores a investigar métodos e técnicas para o desenvolvimento de softwares que atinjam os padrões de qualidade impostos (DELAMARO et al., 2007).

Diferente do *hardware*, o *software* é um artefato lógico; logo, ele não é suscetível à degradação de intempéries. À vista disso, se ele apresentar falhas após sua implantação, certamente, é devido a algum erro que não foi identificado durante o projeto de desenvolvimento do software. Além disso, é necessário enfatizar que o tempo de vida de um *software* não termina após sua implantação (PRESSMAN, 2016). Devido a isso, é comum que as organizações de desenvolvimento despendam cerca de 60 a 70% dos seus recursos financeiros e de pessoal em manutenção de softwares já existentes (PRESSMAN, 2016).

Em resposta a esse problema, a Engenharia de Software - a área da computação cujo objetivo é propor técnicas que apoiam especificação, projeto e evolução de programas (SOMMERVILLE et al., 2011) - estabeleceu princípios sólidos para teste de software, a fim de desenvolvê-lo com alto nível de qualidade e baixo custo. Neste sentido os princípios básicos para testes de software que já estão consolidados pela Engenharia de Software são: o teste é um processo de execução que tem como intenção encontrar erros; um bom caso de teste é aquele que tem a maior probabilidade de encontrar erros que ainda não foram descobertos; e um teste bem sucedido é aquele que encontra erros que ainda não haviam sido detectados dispondo o mínimo de esforço (MYERS, 2011).

Contudo, é importante enfatizar a impossibilidade de realizar todos os possíveis casos de teste, dado que até um programa de pequeno tamanho apresenta um número excepcionalmente grande de casos de teste (DAVIS, 1995). Entretanto, é possível planejar e projetar conjuntos de casos de teste que apresentem um alto grau de cobertura dos critérios de teste sem que, necessariamente, sejam executados todos os casos de teste. Levando isso em consideração, foram estabelecidas diversas técnicas para apoiar a realização de testes de software, entre elas estão: o teste funcional, estrutural e baseada em erros (DELAMARO et al., 2007). Apesar dessas técnicas

serem independentes, são complementares.

De fato, a principal diferença entre elas é a origem da informação utilizada para construir e avaliar os casos de teste. No teste funcional, também conhecido como caixa-preta, o testador não tem acesso ao código-fonte, nesse caso ele assume o papel de usuário final. Nesta técnica de teste, a criação dos casos de testes é orientada de modo que sejam validadas as funcionalidades do sistema (PRESSMAN, 2016). Por outro lado, no teste estrutural, também conhecido como caixa-branca, o código-fonte é fundamental na criação de casos de testes, uma vez que eles são identificados através do conhecimento da estrutura interna do *software* (DELAMARO et al., 2007). Por último, mas não menos importante, a técnica baseada em erros define os critérios de teste a partir de erros típicos cometidos pelos desenvolvedores durante a codificação do software (PRESSMAN, 2016).

Os resultados obtidos por meio da execução das técnicas de teste citadas no parágrafo anterior podem auxiliar na identificação dos casos de testes relevantes para o teste de regressão. O teste de regressão é a reexecução do conjunto de casos de teste da técnica estrutural durante a fase de manutenção do *software* para validar as novas funcionalidades e garantir que o sistema não apresenta nenhum defeito após as alterações realizadas, tendo como referência o código anterior à modificação (MYERS, 2011). Essa reexecução é custosa, quando todos os casos de teste são reexecutados a cada nova versão modificada em teste.

A fim de reduzir o custo de tempo e esforço durante esta reexecução, no trabalho de (ROTHERMEL et al., 2004) foi proposto quatro metodologias que envolvem a reutilização de casos de testes já existentes: (i) Retest All – executar todos os casos de teste que foram desenvolvidos anteriormente para testar a versão modificada; (ii) Seleção – executa de forma seletiva, focando nos subconjuntos dos casos de teste existentes; (iii) Priorização – executa os casos de testes em ordem de prioridade para certificar que os casos de testes mais importantes sejam executados primeiro; (iv) Redução – reduz os futuros testes de regressão através da eliminação permanente de casos de testes redundantes do conjunto casos de testes original.

Apesar desta atividade de teste ser imprescindível para garantir a qualidade do *software*, ela nem sempre é realizada, em razão, de que diversas vezes, não é possível reexecutar os conjuntos de casos de teste toda vez que o *software* sofrer algum tipo de alteração no código original. Portanto, foram propostas algumas abordagens para reduzir o tempo e o custo desse tipo de teste. Uma das abordagens é a utilização da aprendizagem de máquina para apoiar o teste

de regressão (LACHMANN et al., 2016) (NAGAR et al., 2015) (YOO et al., 2009).

Os algoritmos de Aprendizagem de Máquina, um dos subcampos mais relevantes dentro do campo da Inteligência Artificial, procuram aprender um determinado comportamento ou padrão automaticamente dentro de um conjunto de dados, a partir de exemplos ou observações (MITCHELL, 2014). Eles podem ser aplicados para auxiliar a metodologia de redução do conjunto de casos de teste, através da identificação dos casos redundantes que podem ser removidos do conjunto que deve ser executado novamente (LENZ, 2010a) (SANTOS, 2016) (LACHMANN et al., 2016).

O desenvolvimento de uma ferramenta automatizada que realize não apenas a execução de testes, mas também auxilie na redução do conjunto de casos de testes é de grande relevância para desenvolvedores e testadores assegurarem a qualidade do software durante todo o ciclo de vida de desenvolvimento do sistema, especialmente na fase de manutenção. Entretanto, não foi encontrada nas pesquisas de revisão sistemática nenhuma ferramenta similar, apesar de não se poder garantir a inexistência de tal ferramenta.

À vista disso, o objetivo desse trabalho é propor, implementar e validar um método de redução de conjuntos de casos de teste baseado na metodologia proposta por Lenz (LENZ, 2010b). O método proposto faz parte do projeto AI+RTesting, que tem como objetivo apoiar o teste de regressão de métodos escritos na linguagem Java, integrando as ferramentas Jabuti (Teste Estrutural), Mujava (Teste Baseado em Erros) e Weka (Algoritmos de Aprendizado de Máquina). A fim de atingir o objetivo geral, foram estabelecidos os seguintes objetivos específicos:

- Integrar as ferramentas Jabuti e Mujava com o objetivo de capturar os dados de saídas das mesmas e gerar um arquivo ARFF para ser utilizado como entrada para os algoritmos de aprendizagem de máquina da ferramenta Weka.
- Criar uma interface gráfica para a execução da rotina de redução e a visualização dos casos de testes selecionados pela rotina.
- Implementar a validação do conjunto reduzido indicando os percentuais de cobertura e o tempo gasto para execução do novo conjunto de teste.

Neste capítulo foi introduzido o contexto em que o presente trabalho está enquadrado e a apresentação de alguns conceitos básicos de Teste de Software e Aprendizado de Máquina. Além disso, foi estabelecida a motivação em executá-lo e os objetivos necessários para atingi-

lo. O capítulo 2 apresenta detalhadamente os conceitos básicos do Teste de Software que foram introduzidos no capítulo 1. Já o capítulo 3 compreende ainda os princípios relacionados especificamente ao Teste de Regressão e suas metodologias.

O capítulo 4 abrange os conceitos de Aprendizado de Máquina, incluindo a apresentação dos algoritmo de agrupamento e classificação. No capítulo 5 é apresentado o projeto intitulado AI+RTesting, no qual o método será implementado e incorporado. Ademais, são explorados a metodologia proposta, a análise dos requisitos e a descrição do método proposto. Por fim, no capítulo 6 é apresentada uma discussão sobre as questões abordadas nesse trabalho e sugestões de trabalhos futuros.

2 TESTE DE SOFTWARE

Neste capítulo, são apresentados os conceitos fundamentais, as técnicas e os critérios do Teste de Software. Além disso, são abordadas as ferramentas para execução de teste Jabuti e Mujava.

2.1 CONCEITOS BÁSICOS

O teste de software, uma das áreas de pesquisa da Engenharia de Software, consiste em analisar um programa com a intenção de detectar o máximo de defeitos, dispondo o mínimo de esforço (MYERS, 2011). Para Myers, atividade de teste é realizada através da execução do programa fornecendo os dados necessários para a execução. Posteriormente, é comparada a saída gerada com o resultado esperado do caso executado. Nessa perspectiva, o caso de teste é formado pelo dado fornecido e sua saída esperada (PRESSMAN, 2016).

Segundo Sommerville(SOMMERVILLE et al., 2011), o processo de teste tem dois objetivos distintos: o primeiro é demonstrar ao desenvolvedor e ao cliente que o software atende a seus requisitos. O testador elabora um determinado conjunto de casos de teste que espelham o comportamento esperado do sistema. Já o segundo é detectar os cenários em que o *software* não comportou-se em conformidade com as especificações do sistema. Nesse caso, o testador prepara o conjunto de casos de teste a externar defeitos. É importante enfatizar que os testes podem mostrar apenas remover a presença de defeitos, e não sua ausência (DIJKSTRA, 1972).

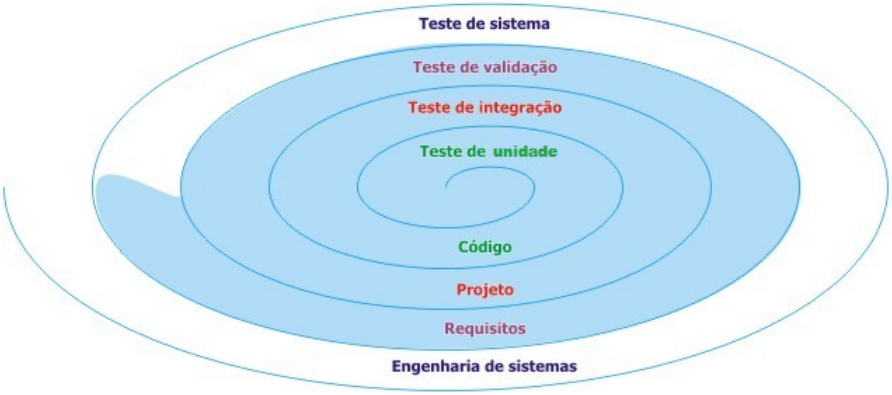
Diante disso, é indispensável que a atividade de teste seja realizada de forma estratégica e sistemática, caso contrário, tempo é desperdiçado, esforço desnecessário é empregado, e pior ainda, erros críticos podem não serem detectados. Para garantir uma estratégia de teste efetiva, é necessário elaborar planos e procedimentos de teste que, consequentemente, levarão à descoberta de erros em cada etapa do processo de desenvolvimento do *software* (PRESSMAN, 2016).

2.2 ESTRATÉGIA DE TESTE

De acordo com Pressman(PRESSMAN, 2016), a estratégia para testes de software fornece uma espécie de roteiro descrevendo cada etapa a ser executada durante a atividade de

teste. Ele sugere as quatro fases que devem compor a estratégia de teste, sendo que estas podem ser apresentadas no formato espiral, conforme ilustrado na Figura 2.1:

Figura 2.1 – Estratégia de Teste



Fonte – Pressman (2016)

Em primeiro lugar, o teste de unidade é executado para identificar erros de lógica e de implementação em cada módulo do software, separadamente. Logo após, é aplicado o teste de integração visando a identificação de erros associados às interfaces entre os módulos durante a integração da estrutura do programa. Posteriormente, é realizado o teste de validação com o objetivo de verificar se o *software* funciona de acordo com a especificação dos requisitos do sistema. No fim, o teste de sistema é realizado após a integração do sistema para identificar erros de funções e características de desempenho que não estejam de acordo com o esperado (PRESSMAN, 2005) (SOMMERVILLE et al., 2011) (DELAMARO et al., 2007).

2.3 TÉCNICAS E CRITÉRIOS DE TESTE

Em geral, é impraticável, muitas vezes até impossível, encontrar todos os erros em um programa. Com o propósito de mitigar esse problema, é necessário garantir que a atividade de teste seja realizada de modo sistemático e teoricamente embasado (MYERS, 2011). Para garantir isso, são aplicadas técnicas e critérios de teste que ajudam o testador e o desenvolvedor a avaliar a maneira em que os casos de teste serão projetados. Nesse contexto, os critérios de teste podem ser utilizados tanto para auxiliar na geração de conjunto de casos de teste como para auxiliar na avaliação da adequação desses conjuntos (DELAMARO et al., 2007).

Por via de regra, os critérios de teste de software são estipulados baseado em três técnicas: a funcional, a estrutural e a baseada em erros. Essa técnicas de teste se diferenciam

pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste, lembrando que essas técnicas de teste são complementares (DELAMARO et al., 2007).

2.3.1 Técnica Funcional

O teste funcional, também conhecido como caixa-preta, concentra-se apenas em encontrar as situações em que o *software* não se comporta de acordo com a sua especificação, sem levar em consideração o comportamento interno e a estrutura do programa (MYERS, 2011) (SOMMERVILLE et al., 2011). Sendo assim, essa técnica de teste tem como principal objetivo determinar se o programa satisfaz aos requisitos funcionais que foram especificados.

De acordo com Pressman, o teste funcional é dividido em duas fases. Primeiramente, é identificado as funcionalidades que devem ser desempenhadas pelo *software*. Logo em seguida, são criados os casos de teste para verificar se estas funcionalidades estão sendo efetuadas corretamente (PRESSMAN, 2016). Após a realização da atividade de teste, os dados de entrada e de saídas são comparados com o resultado esperado durante a especificação dos casos de teste (NETO, 2007). Se estiverem idênticos as funcionalidades estão corretas, caso contrário, a atividade de teste identificou erros no *software*.

É importante enfatizar que estas funcionalidades são obtidas através da especificação do *software*, portanto, é fundamental que esta seja elaborada meticulosamente. Entretanto, o problema do teste funcional é que, geralmente, esta especificação é informal (DELAMARO et al., 2007). Apesar dessa desvantagem, o teste funcional pode ser utilizado em qualquer contexto (procedimental ou orientado a objetos) e em qualquer fase de teste, sem a necessidade de modificação (DELAMARO et al., 2007).

Como já foi dito anteriormente, existe uma grande dificuldade em realizar testes exaustivos. Uma das formas de combater este problema é a utilização de um dos principais critérios desta técnica de teste que é denominado particionamento em classes de equivalência.

2.3.1.1 Particionamento em Classes de Equivalência

Considerando que ao testar um programa, o testador está limitado a tentar subconjuntos de todas as entradas possíveis, é necessário identificar quais são os subconjuntos com a maior probabilidade de encontrar a maioria dos erros (MYERS, 2011). Sendo assim, a proposta

deste critério é agrupar as entradas do programa em classes de equivalência válidas e inválidas, onde cada classe é especificada de acordo com o tipo de erro que deseja-se verificar (ROCHA et al., 2001) (DELAMARO et al., 2007).

Assim, é selecionado o menor número possível de casos de teste, respaldado no pressuposto que um elemento de uma determinada classe representaria toda a classe (PRESSMAN, 2016) (SOMMERVILLE et al., 2011). A utilização do particionamento proporciona a avaliação dos requisitos de forma sistemática e limita o número de casos de teste existentes (DELAMARO et al., 2007).

2.3.2 Técnica Estrutural

A técnica estrutural, também conhecida como caixa-branca, visa analisar a lógica interna do software. Esse técnica é aplicada juntamente com a técnica funcional, uma vez que os dados obtidos pela aplicação dessas técnicas são muito pertinentes nas atividades de manutenção do *software* (DELAMARO et al., 2007). Contudo, em oposição ao teste funcional, no teste estrutural, o código-fonte é fundamental na criação de casos de testes, uma vez que eles são identificados através do conhecimento da estrutura interna do *software* (PRESSMAN, 2016) (SOMMERVILLE et al., 2011).

O teste estrutural tem como propósito analisar o código-fonte e selecionar um conjunto de casos de teste que exercitem partes do código-fonte e não de sua especificação. Para garantir que o teste estrutural exercite todas as partes do código, os casos de teste são elaborados de forma que todos os possíveis caminhos do fluxo de controle sejam testados pelo menos uma vez (MYERS, 2011). Com intuito de identificar o fluxo de controle, um determinado programa deve ser decomposto em um conjunto de blocos disjuntos de comandos, no qual a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada (DELAMARO et al., 2007). Em geral, esses caminhos são representados através de grafo de fluxo de controle (VINCENZI, 2004).

2.3.2.1 Critério Baseado em Fluxo de Controle

Nos critérios baseados em fluxo de controle são aplicados as características de controle da execução do programa, como comandos ou desvios, para definir quais são os requisitos de teste. Estes critérios de teste são crucias, dado que eles viabilizam um mecanismo sistemático

para especificar e avaliar a eficiência do conjunto de casos de teste definido (VINCENZI, 2004).

No teste de fluxo de controle existem dois principais critérios: todos-nós e todos-arcos (arestas) (MALDONADO et al., 2007) (MYERS et al., 2011). Na programação orientada a objetos, seja T um conjunto de casos de teste para um programa P, *DUG* seria o grafo de fluxo de controle correspondente de P, sendo $\prod$ o conjunto de caminhos executados por T. Um nó i está incluído em $\prod$, se $\prod$ contém um caminho $(n_i, ..., n_m)$, tal que $i = n_j$ para algum j, $1 \leq j \leq m$. Da mesma maneira, um arco (i_1, i_2) é incluso em $\prod$, caso $\prod$ contém um caminho $(n_i, ..., n_m)$ tal que $i_1 = n_j$ e $i_2 = n_{j+1}$ para algum j, $1 \leq j \leq m-1$. Um caminho $(i_1 ... i_k)$ está incluído em $\prod$ se $\prod$ contém um caminho $(n_i, ..., n_m)$ e $i_1 = n_j$, $i_2 = n_j + 1$,...,$i_k = n_{jk+1}$ para algum j, $1 \leq j \leq m-k+1$ (VINCENZI, 2004). Os critérios todos-nós e todos-arcos podem ser definidos da seguinte forma:

• Critério Todos-Nós: demanda que o programa executado passe, pelo menos uma vez, em cada vértice do grafo de fluxo, ou seja, cada comando do programa deve ser executado pelo menos uma vez (MALDONADO et al., 2007). $\prod$ satisfaz o critério todos-nós se cada nó $n \in N$ de um grafo *DUG* está incluso em $\prod$, garantindo que todas as instruções (comandos) de um determinado método tenham sido executadas ao menos uma vez por algum caso de teste T (VINCENZI, 2004).

• Critério Todos-Arcos: demanda que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez (MALDONADO et al., 2007). $\prod$ satisfaz o critério todos-arcos se cada arco $e \in E$ de um grafo *DUG* está incluso em $\prod$ (VINCENZI, 2004). Este critério garante que cada arco de um grafo *DUG* seja exercitado pelo menos uma vez por algum caso de teste T.

2.3.2.2 Critério Baseado em Fluxo de Dados

Além dos critérios de fluxo de controle, existem também os critérios baseados em fluxo de dados, que utilizam as informações do fluxo de dados do programa para estipular os requisitos de teste (DELAMARO et al., 2007). Neste critério, é necessário testar todas as interações relacionadas a definições de variáveis e suas referências subsequentes (MAFRA et al., 2009). Os critérios de fluxo de dados compreendem: todos-usos e todos-potenciais-usos. Sendo que o critério todos-usos englobam os sub-critérios: todos-c-usos(todos usos computacionais) e todos-p-usos(todos usos predicativos). Os critérios todos-usos e todos-potencias-usos são

definidos da seguinte maneira:

• Critério Todos-Usos: no qual todas as associações entre uma definição de variável e seus subsequentes usos sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida (MALDONADO et al., 2007). $\prod$ satisfaz o critério todos-usos se cada nó $i \in N$ e para todo $x \in def(i)$, inclui um caminho livre de definição em relação a x do nó i para todos os elementos $dcu(x,i)$ e $dpu(x,i)$. Portanto, neste critério toda associação definição-uso $(i,j,x)\,|\,j \in dcu(x,i)$ e $(i,\ (j,k),x)\,|\,(j,k) \in dpu(x,i)$ deve ser exercitada pelo menos uma vez por algum caso de teste T (VINCENZI, 2004).

• Critério Todos-Potenciais-Usos: $\prod$ satisfaz o critério todos-potenciais-usos se cada nó $i \in N$ e para todo $x \in defg(i)$, inclui um caminho livre de definição em relação a x do nó i para todos os elementos $pdcu(x,i)$ e $pdpu(x,i)$. Ou seja, este critério requer que toda potencial-associação definição-uso $(i,j,x)\,|\,j \in pdcu(x,i)$ e $(i,\ (j,k),x)\,|\,(j,k) \in pdpu(x,i)$, seja exercitada pelo menos uma vez por algum caso de teste T (VINCENZI, 2004).

2.3.2.3 Ferramenta Jabuti

Para a execução do teste unitário estrutural, será utilizada a ferramenta Jabuti (Java Bytecode Understanding and Testing). Esta ferramenta de código aberto desenvolvida por Vincenzi (2004) é utilizada para executar a técnica estrutural em programas na linguagem Java. O teste é executado sobre o código objeto em Java e permite que os requisitos de teste de cada um de seus critérios sejam visualizados no código.

A ferramenta executa tanto a estratégia de teste baseado em fluxo de dados quanto a fluxo de controle. Na Jabuti, é possível importar os casos de teste do JUnit, acompanhar e visualizar a cobertura dos critérios. Além disso, oferece um conjunto de métricas estáticas para avaliar a complexidade das classes que compõem o programa ou o componente, e implementa heurísticas de particionamento de programas que facilitam a localização de defeitos (ASCARI, 2009).

2.3.3 Técnica Baseada em Erros

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais frequentes no processo de desenvolvimento de software para derivar os casos de teste. Esta técnica enfatiza os erros que o programador ou projetista pode cometer durante o desenvolvimento. O objetivo da técnica baseada em erros é mostrar a ausência de tais defeitos no programa e o principal critério de teste aplicável a esta técnica é Análise de Mutantes (MALDONADO et al., 2007).

2.3.3.1 Critério de Análise de Mutantes

Outra técnica de teste amplamente aplicada é o critério Análise de Mutantes. Nessa técnica é utilizado um conjunto de programas transmutados obtidos a partir de um determinado programa P para ponderar se um caso de teste T é apropriado para o teste de P. O objetivo desse critério é estabelecer um conjunto de casos de teste que consiga expor, através da execução de P, as discrepâncias de comportamentos que existem entre programa P e seus mutantes (OLIVEIRA et al., 2013).

Esse critério está fundamentado no pressuposto conhecido como desenvolvedor competente, admitindo-se que um desenvolvedor codificou a lógica do programa. Considerando que este pressuposto é verossímil, então é deduzido que os erros introduzidos no programa P são apenas pequenos desvios sintáticos, que apesar de não terem sido identificados pelo compilador como erros sintáticos, modificam a semântica do programa P e, como resultado, acarreta um comportamento incorreto no programa P (DEMILLO et al., 1978) (DELAMARO et al., 2007).

Primeiramente, é executado o conjunto de casos de teste no programa P original. Caso os resultados obtidos estejam em conformidade com os resultados esperados, esse conjunto de casos de teste é executado nos programas mutantes para identificar se os casos de teste identificam os erros do programa P transmutado. Nesse contexto, os mutantes são erros inseridos propositalmente no programa P original. Para diferenciar o comportamento do programa original e do programa mutante deve ser gerado um dado de teste. Se houver uma diferença, o mutante é declarado como morto e concluí-se que o programa original está livre do erro apontado por esse mutante (OLIVEIRA et al., 2013).

Caso contrário, o mutante declarado como vivo e o mesmo deve ser analisado

manualmente para determinar se o mutante vivo é um erro no programa P ou se representa um mutante equivalente ao programa P. Se o mutante for considerado equivalente, é declarado que o programa P está correto ou existe chances mínimas de que aquele determinado erro ocorra. Se o mutante vivo não for equivalente, é necessário gerar novos casos de teste para detectar as discordâncias entre o programa original e o mutante (MALDONADO et al., 2007).

Com a finalidade de qualificar a eficiência do conjunto de casos de teste, foi estipulado o escore de mutação. O escore é a medida objetiva utilizada para identificar a cobertura do conjunto de casos de teste e pode variar entre 0 e 1. Nessa conjuntura, quanto maior o score de mutação, mais adequado é o conjunto de casos de teste para o programa P (MALDONADO et al., 2007) (OLIVEIRA et al., 2013). O escore de mutação é calculado da seguinte maneira:

$$ms(P,T) = \frac{\mathrm{DM(P,T)}}{\mathrm{M(P)} - EM(P)} \tag{2.1}$$

Sendo:

•$DM(P,T)$: número de mutantes mortos pelos casos de teste em T;

•$M(P)$: número total de mutantes gerados;

•$EM(P)$: número de mutantes gerados equivalentes a P.

2.3.3.2 Ferramenta Mujava

A execução dessa técnica de teste será realizada na ferramenta de código aberto Mujava que foi desenvolvida nas universidades KAIST (Instituto Avançado de Ciência e Tecnologia da Coreia) e Universidade George Mason. Essa ferramenta tem como objetivo apoiar a execução da técnica de teste baseada em erros e também dar suporte a programas implementados na linguagem Java. Na Mujava é possível realizar testes automatizados e consequentemente, viabilizar a redução do ciclo de testes e aumentar a cobertura do software.

Assim como a Jabuti, esta ferramenta trabalha sobre o código objetivo e requer duas compilações: compilação do código fonte e dos mutantes gerados. Assim, para um programa P, cada mutante de P é decorrente de um resultado de uma única modificação em P. Sendo assim, cada mutante de P se diferencia do programa original apenas em uma declaração mutante. A

forma como estas declarações são modificadas é definida a partir de operadores de mutação e cada um deles corresponde uma classe de erros comuns cometidos pelos programadores. (MA; OFFUTT, 2005).

2.4 CONSIDERAÇÕES FINAIS

Este capítulo apresentou a fundamentação teórica da atividade de teste de software. Inicialmente, foram abordados os conceitos fundamentais e as estratégias utilizadas para realização de testes. Logo após, foram explanadas as técnicas, os critérios e as ferramentas que podem ser aplicadas nessa atividade. No próximo capítulo, serão expostos os conceitos básicos do teste de regressão e as técnicas mais utilizadas pra executa-lo.

3 TESTE DE REGRESSÃO

Neste capítulo, são apresentados os conceitos básicos do Teste de Regressão e as metodologias utilizadas para a realização do mesmo. Além disso, foram descritos os trabalhos que relacionam-se com o presente trabalho.

3.1 CONCEITOS BÁSICOS

O teste de regressão é o processo de validação de um software que foi modificado para detectar se novos erros foram introduzidos no código previamente testado e fornecer confiança de que as modificações estão corretas. Por via de regra, o *software* (PRESSMAN, 2016) (ROTHERMEL et al., 2004) pode sofrer quatro tipos de modificações durante a fase de manutenção: adaptativa - modificações são realizadas para que o software se adapte a um novo ambiente, corretiva - modificações são realizadas com o propósito de corrigir erros, preventiva - modificações são realizadas visando facilitar futuras manutenções, e evolutiva - modificações são realizadas para adicionar novas funcionalidades (DELAMARO et al., 2007).

Independentemente do tipo da modificação realizada no *software*, é imprescindível retestar novamente toda a aplicação para garantir que nenhum efeito colateral foi inadvertidamente inserido no *software*. Não obstante, em diversos casos a atividade do teste de regressão é negligenciada ou realizada sem embasamento em nenhum tipo de metodologia. Isso se dá pelo fato de que essa atividade de teste é significativamente custosa, tanto em tempo quanto em recursos humanos (ROTHERMEL et al., 2004).

Como citado anteriormente, muitas vezes o teste de regressão é executado sem nenhuma base teórica e por esta razão a estratégia de teste de regressão comumente utilizada é o *retest all*. No *retest all*, todo conjunto de casos de teste original é reexecutado a cada alteração implementada no *software* (GRAVES et al., 2001). Todavia, essa estratégia é a mais inviável, em função de que consume mais tempo e recursos do que todas as outras estratégias de teste (ROTHERMEL et al., 2004).

Tendo em vista esse problema, diversos pesquisadores propuseram técnicas que de alguma forma reduzem esse custo, entre esses pesquisadores está o cientista da computação Gregg Rothermel. Ele estabeleceu três metodologias para tentar limitar o número de casos de teste que devem ser reexecutados durante o teste de regressão, essas metodologias são

denominadas redução, seleção e priorização (ROTHERMEL et al., 2004).

3.2 METODOLOGIAS DO TESTE DE REGRESSÃO

A seleção de casos de teste consiste em selecionar um subconjunto do conjunto de casos de teste original baseando-se em algum tipo critério que o testador tem maior interesse (MAFRA et al., 2009). Esses critérios de seleção podem ser a cobertura de teste, as funcionalidades mais críticas ou frequentemente utilizadas, o tempo de execução (SIMONS, 2010).

Após a definição desses critérios, é identificado no conjunto de casos de testes original quais são os casos de testes que tem o propósito de testar aquele determinado critério. Sendo assim, somente os casos de teste que exercitam essas partes deverão ser reexecutados. Assim reduzindo consideravelmente o conjunto de testes a ser aplicado durante o teste de regressão (MAFRA et al., 2009).

Já na metodologia de priorização de casos de teste (KUKOLJ et al., 2013) (SIMONS, 2010) são estabelecidos critérios para ordenar a execução dos casos de teste. Esses critérios podem ser os mesmos critérios estabelecidos na metodologia de seleção previamente mencionados na seção anterior.

O principal objetivo dessa metodologia é atingir uma alta cobertura do código através da detecção de erros o mais rápido possível. Além disso, também é possível garantir que as funcionalidades com maior grau de importância para o testador estejam funcionando corretamente, já que essa metodologia permite criar uma ordem de precedência.

Outro ponto importante é que a priorização não elimina casos de teste, e assim, reduz o risco de perda na cobertura e minimiza o risco de áreas importantes do *software* não seja testada.

Por outro lado, na metodologia de redução de conjuntos de casos de teste são removidos permanentemente os casos de teste redundantes do conjunto de casos de teste original (ROTHERMEL et al., 2004). Segundo Rothermel (ROTHERMEL et al., 2004), essa redundância é inserida no conjunto de casos de teste durante a fase de manutenção do programa, dado que novos casos de teste são adicionados no conjunto para validar as novas funcionalidades implementadas no *software*. Logo, a cada alteração sofrida pelo *software*, a tendência é que o

número de casos de teste no conjunto de casos de teste aumente.

Diante disso, o objetivo principal da redução do caso de teste é diminuir o número de casos de teste sem perdas na cobertura dos casos de teste, a fim de minimizar o tempo e custo de execução (SAIFAN et al., 2016). Existem várias técnicas que podem ser usadas para reduzir casos de teste, incluindo a utilização de algoritmos de agrupamento e classificação, como K-means e J48, respectivamente. Essas técnicas serão detalhadas mais adiante nesse presente trabalho.

A redução de casos de teste difere da seleção de casos de teste. Na redução, os casos de teste redundantes são permanentemente removidos do conjunto original (JONES; HARROLD, 2003). Já na seleção, é selecionado um subconjunto de casos de teste para ser executado sobre uma determinada versão do software (MAFRA et al., 2009). Dessa forma, pode-se observar que as três metodologias são complementares, podendo ser utilizadas em conjunto ou separadamente.

3.3 CONSIDERAÇÕES FINAIS

Este capítulo relatou os principais conceitos e as metodologias referentes ao teste de regressão. No próximo capítulo, serão explanados os conceitos, os sistemas e os algoritmos pertinentes para a compreensão do aprendizado de máquina. Ademais, será apresentado a ferramenta Weka que é muito utilizada para a realização de experimentos em aprendizado de máquina.

4 APRENDIZADO DE MÁQUINA

Neste capítulo, são explicados os principais conceitos de Aprendizado de Máquina com ênfase nas técnicas de agrupamento e classificação. Em seguida, será apresentada a ferramenta Weka.

4.1 CONCEITOS BÁSICOS

De acordo com Mitchell (MITCHELL, 2014), o Aprendizado de Máquina, um dos subcampos da Inteligência Artificial, estabeleceu um conjunto de métodos, técnicas e ferramentas cujo o objetivo principal é fornecer à máquina a capacidade de aprender um determinado comportamento ou padrão automaticamente dentro de um conjunto de dados a partir de exemplos ou observações, assim melhorando seu desempenho por meio da experiência. Tendo dito isto, pode-se concluir que o sistema de aprendizado é um programa de computador que toma decisões baseado em experiências acumuladas através da solução bem sucedida de problemas anteriores.

Existem basicamente dois tipos de sistema de aprendizado de máquina: o supervisionado e o não supervisionado. Nos sistemas não supervisionados, os algoritmos de indução analisam os exemplos fornecidos e procuram determinar como eles serão agrupados para formar os *clusters*. Após esse processo de agrupamento, em geral, é analisado o significado de cada *clusters* com base no contexto do problema analisado. Lembrando que o aprendizado não supervisionado tem a finalidade de identificar os padrões existentes nos dados e organizar os mesmos em *clusters* e, por isso, é possível detectar similaridades e diferenças entre os padrões existentes (MONARD; BARANAUSKAS, 2003) (WITTEN et al., 2016). Para o desenvolvimento deste presente trabalho será utilizado este sistema através dos algoritmos de agrupamento, no qual será identificado automaticamente padrões nos conjuntos de dados coletados após a execução do conjunto de casos de teste.

Já nos sistemas supervisionados, é fornecido para o algoritmo de aprendizado um conjunto de exemplos de treinamento, nos quais os rótulos da classe associada são conhecidos. Para cada exemplo de treinamento, é descrito um vetor de valores de características e um rótulo da classe associada. O algoritmo de indução extrai um classificador a partir de um conjunto de exemplos rotulados para determinar corretamente classes de novos exemplos que ainda não tenham o rótulo da classe (MONARD; BARANAUSKAS, 2003) (WITTEN et al., 2016). Neste

trabalho, o sistema supervisionado será aplicado através dos algoritmos de classificação para associar cada caso de teste com seu respectivo *cluster*, e com isso, gerar regras de classificação sobre o conjunto de exemplos obtidos com as técnicas de agrupamento.

4.2 ALGORITMOS DE AGRUPAMENTO

O agrupamento, que também é denominado *cluster*, é uma coleção de registros semelhantes entre si, por outro lado, são diferentes dos demais registros agrupamentos (HAN et al., 2011). Esses registros são agrupados de acordo com a similaridade que existe dentro de uma população de registros definidos, baseando-se nas características que estes objetos possuem (SIMONS, 2010) (WITTEN et al., 2016).

Essas técnicas de agrupamento são definidas como não supervisionadas, visto que os rótulos das classes dos dados de treinamento que estão sendo agrupados não estão definidos. Em contrapartida, diferenciando-se da técnica de classificação, na qual os rótulos das classes são conhecidos e os dados de treinamento estão devidamente classificados.

Nos algoritmos de agrupamento existe uma coesão em relação aos mecanismos genéricos aplicados como critérios de tipificação. Dessa maneira, eles podem auxiliar na escolha daquele que melhor se ajusta ao conjunto de dados a serem agrupados. Contudo, os algoritmos de sistemas não supervisionada podem ser utilizados na solução de qualquer problema de agrupamento (WITTEN et al., 2016).

Para realizar o agrupamento dos dados neste trabalho, foram aplicados os seguintes algoritmos: o EM, Hierárquico e K-means. O algoritmo EM é um método iterativo que estima parâmetros que sejam os mais consistentes com os dados do conjunto fornecido no sentido de maximizar a função de verossimilhança (MCLACHLAN; KRISHNAN, 2008). Já o algoritmo hierárquico organiza o conjunto de dados em uma estrutura hierárquica de acordo com a proximidade entre os indivíduos (MITCHELL, 2014).

O K-Means, por sua vez, é utilizado para classificar informações com base nos próprios dados sem a necessidade de uma pré-classificação existente. Este algoritmo faz uma análise a partir das comparações entre os valores numéricos de todos os dados para criar classificações e, posteriormente, indicar a qual classe (*cluster*) cada dado pertence (DASH; DASH, 2012).

4.3 ALGORITMOS DE CLASSIFICAÇÃO

Os algoritmos de classificação tem como finalidade identificar um conjunto de modelos que descrevem e distinguem classes e utilizar o modelo refinado para predizer classes de registros que ainda não foram classificados (WITTEN et al., 2016). Cada classe é correspondente de um padrão de valores dos atributos previsores. Sendo assim, é possível encontrar a relação entre os atributos previsores e as classes (FREITAS; LAVINGTON, 1997).

Para isso, alguns algoritmos de classificação geram árvores de decisão para predizerem as classes de acordo com os valores dos atributos de um conjunto de dados. A estrutura da árvore é organizada da seguinte maneira: nó interno, que é rotulado com o nome de um dos atributos, ramos (arestas) saindo de um nó interno são rotulados com valores do atributo naquele nó e folha que é rotulada com uma classe, a qual é a classe prevista para exemplos que pertençam aquele nó folha (CARVALHO, 2005). O propósito de uma árvore de decisão é sistematizar os dados, assim facilitando o processo de tomada de decisão.

Um dos exemplos de algoritmos de classificação que constroem árvores de decisão é o J48. O algoritmo J48 tem como objetivo gerar uma árvore de decisão baseada em um conjunto de dados de treinamento, sendo este modelo utilizado para classificar as instâncias do conjunto de dados de teste. Durante o processo de aplicação do algoritmo J48 é importante conhecer quais parâmetros podem ser alterados com o propósito de atingir melhores resultados. Esses parâmetros são o número mínimo de instâncias por folha, a construção de uma árvore binária, entre outros (OLIVEIRA et al., 2013).

O algoritmo J48 é amplamente utilizado nas pesquisas que envolvem mineração de dados, em razão de que este algoritmo é adequado para os procedimentos que envolvem variáveis qualitativas contínuas e discretas em base de dados. O J48 gera como resultado regras de classificação em formato de árvores de decisão, no qual a sua estrutura permite que estas sejam interpretadas e armazenadas no formato da estrutura de dados árvore. Nesse trabalho, o J48 foi aplicado no processo de classificação das instâncias do conjunto de dados de teste.

4.4 FERRAMENTA WEKA

Uma das ferramentas mais conhecidas para manipular os algoritmos de aprendizado de máquina é a Weka. A ferramenta Weka, desenvolvida na Universidade de Waikato, é um

pacote de *software* composto por algoritmos de aprendizado de máquina para realizar operações de mineração de dados e validação/verificação dos resultados (HALL et al., 2009). Os algoritmos de aprendizagem de máquina que fazem parte do Weka são aplicados para pré-processamento de dados, classificação, regressão, agrupamento, regras de associação e visualização.

A ferramenta é implementada na linguagem Java e fornece uma API flexível que permite a integração de suas classes dentro de programas Java. Ao adicionar a biblioteca .jar do Weka ao projeto Java é possível ter acesso às classes dos algoritmos de aprendizado de máquina e a maioria de suas funções através de seus pacotes como: remover atributos, selecionar a fonte de dados e conduzir o processo de mineração de dados, realizando avaliação dos resultados obtidos e a comparação de algoritmos (HALL et al., 2009).

4.5 CONSIDERAÇÕES FINAIS

Este capítulo abordou os conceitos básicos e as técnicas relacionadas ao aprendizado de máquina, sendo que foi explicado com maior detalhamento as técnicas de agrupamento e classificação. No próximo capitulo, serão explicados a utilização dessas técnicas para auxiliar na identificação de classes de equivalência dos dados de execução de casos de teste das técnicas estrutural e baseada em erros, e a geração das regras de classificação dos dados obtidos no agrupamento. Posteriormente, essas informações serão utilizadas na realização do método de redução.

5 MÉTODO PARA REMOÇÃO DE REDUNDÂNCIAS DE CONJUNTOS DE CASOS DE TESTE

Neste capítulo, serão apresentados o projeto de pesquisa no qual o trabalho está inserido, as etapas do projeto, a metodologia aplicada para a implementação e, principalmente, o detalhamento do método de redução proposto.

5.1 PROJETO AI+RTESTING

O Projeto AI+RTesting - Uma Metodologia para Testes de Regressão Aplicando Técnicas da Inteligência Artificial é um projeto de pesquisa realizado por alunos de graduação do curso de Sistemas de Informação da Universidade do Estado da Bahia - Campus I e que tem como orientador o M.Sc. Alexandre Rafael Lenz. Esse projeto tem como finalidade o desenvolvimento de uma ferramenta automatizada para apoiar o teste de regressão.

A motivação deste projeto foi a ausência de uma ferramenta automatizada que realizasse não apenas a execução de testes, mas também auxiliasse na execução das metodologias do teste de regressão proposto por (ROTHERMEL et al., 2004): a redução, a seleção e a priorização de casos de teste. Essa ferramenta é de grande relevância para desenvolvedores e testadores assegurarem a qualidade do software durante todo o ciclo de vida de desenvolvimento do sistema, especialmente na fase de manutenção.

O projeto propõe dar suporte para a execução do teste unitário funcional, estrutural e baseado em erros, para métodos escritos na linguagem Java. Além disso, auxiliará na execução das três metodologias do teste de regressão através da aplicação de técnicas de aprendizado de máquina. O presente projeto contemplará análise, implementação e validação do método responsável pela redução dos casos de teste. Para o desenvolvimento dessa ferramenta, é necessário a integração de ferramentas para execução das atividades de teste e execução dos algoritmos de agrupamento e classificação, entre elas estão: Jabuti, Mujava e Weka.

5.2 TRABALHOS RELACIONADOS

Em um trabalho publicado por (LENZ, 2010b), foi apresentada uma abordagem para apoiar o teste de regressão em programas implementados na linguagem C, utilizando Aprendizagem de Máquina para identificar classes de equivalência de um programa e a partir

disso gerar regras para auxiliar na redução do conjunto de casos de teste. Observa-se em outro trabalho que essa metodologia foi igualmente aplicada em programas implementados na linguagem Java e foram obtidos bons resultados (SANTOS, 2016). Nesse trabalho foram propostos alguns direcionamentos de como aprimorar essa abordagem, tendo como a ação principal automatização deste processo de teste.

A Tabela 5.1 apresenta um comparativo dos trabalhos relacionados com o trabalho proposto.

Tabela 5.1 – Comparativo de trabalhos relacionados.

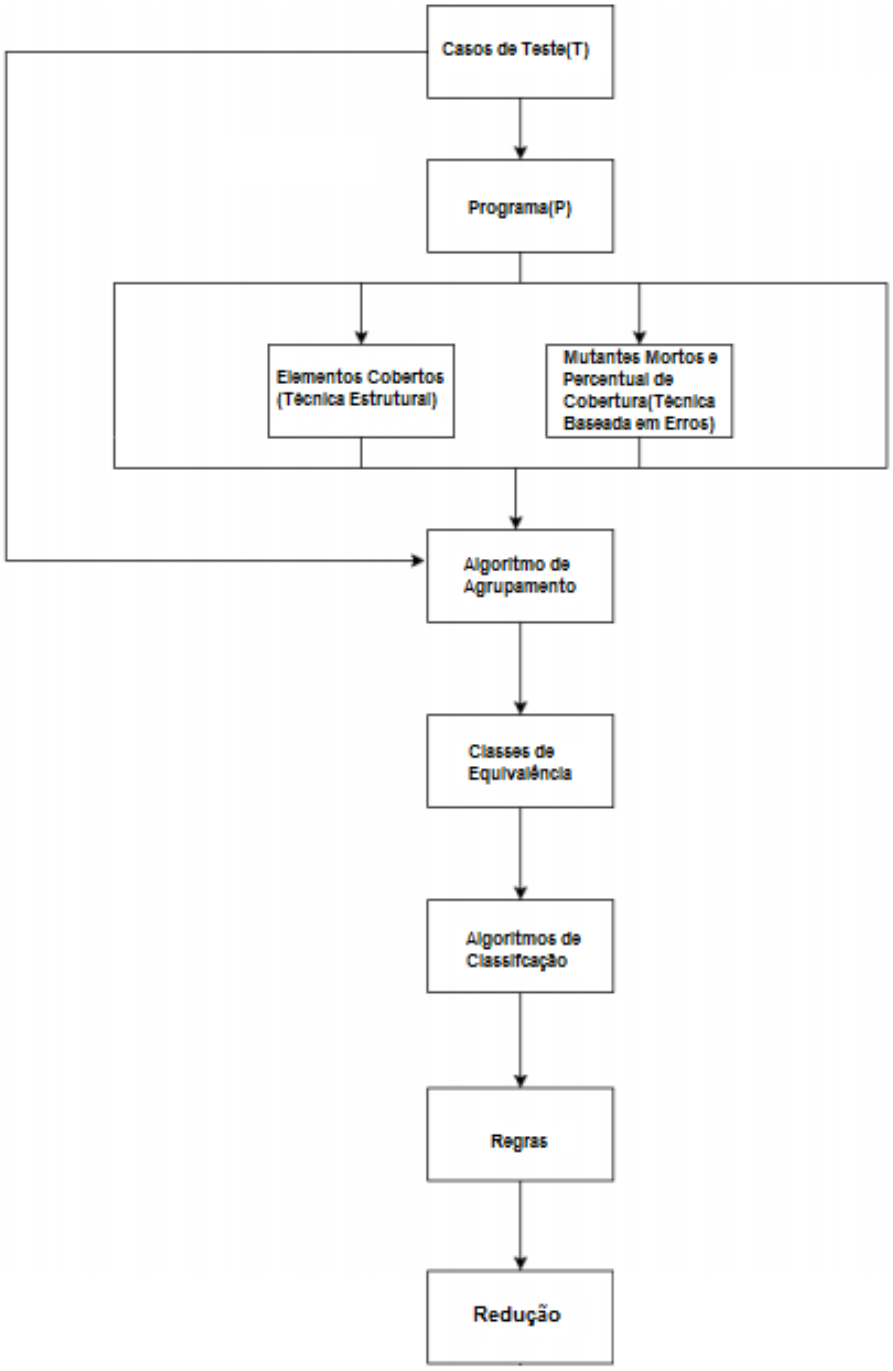
Trabalhos	**Linguagem**	**Agrupamento**	**Classificação**	**Dados Analisados**	**Metodologia**
Lenz (2010a)	C	K-means Cobweb EM	J48	Casos de Teste (entradas e saídas) Análise de mutantes Critérios estruturais	Seleção Priorização Redução
Muthayla e Naidu (2011)	C	K-means	Outros	Casos de Teste (entradas e saídas)	Redução
Santos (2016)	Java	K-means Hierárquico EM DBSCAN	J48	Casos de Teste (entradas e saídas) Análise de mutantes Critérios estruturais	Redução
Cristo (2017)	Java	K-means Hierárquico EM	J48	Critérios estruturais Análise de mutantes	Seleção Priorização
Santos (2018)	Java	K-means Hierárquico EM	J48	Análise de mutantes	Redução (Progressão Geométrica)
Trabalho Proposto	Java	K-means Hierárquico EM	J48	Critérios estruturais Análise de mutantes	Redução (Progressão Aritmética)

5.3 METODOLOGIA

O propósito desse trabalho é implementar um método para remover os casos de teste redundantes do conjuntos de casos de teste para programas implementados na linguagem java. Com intuito de realizar esse processo de forma sistemática e com embasamento teórico será aplicada o *workflow* proposto por (LENZ, 2010b).

A Figura 5.1, apresenta uma adaptação do fluxograma do *workflow* proposto Lenz (2010b).

Figura 5.1 – Representação gráfica do *workflow*



Fonte – Lenz (2010b), adaptada.

Conforme apresentado na figura 5.1, primeiramente um conjunto de casos de teste T

é definido para testar um programa P. Posteriormente, T é executado em uma ferramenta Jabuti, retornando o percentual de cobertura dos critérios da técnica estrutural (todos-nós, todos-arcos, todos-usos e todos-potenciais-usos).

Após a execução de teste estrutural e e do teste baseado em erros, são coletados os resultados obtidos para preencher o arquivo que será executado na ferramenta Weka (SANTOS, 2018). Nesse momento, os algoritmos de agrupamento são aplicados sobre o conjunto de dados obtidos. Portanto, é possível identificar as classes de equivalência do programa em teste P, e os casos de teste T associados a cada classe. Logo em seguida, essas classes são utilizadas como dado de entrada do algoritmo de classificação (LENZ, 2010b). O algoritmo de classificação, por sua vez, gera regras de classificação. Estas regras são interpretadas e posteriormente é aplicado o método proposto que realiza a redução dos casos de teste.

Para o desenvolvimento do método foram estabelecidas as etapas que compõem a metodologia: análise de requisitos, projeto, desenvolvimento e validação. Na etapa de análise de requisitos foram definidos: análise das ferramentas Jabuti e Mujava, a definição dos requisitos do projeto AI+RTesting, análise da estrutura do arquivo Arrf, e a definição dos objetivos, das restrições e das informações referentes aos requisitos do método. Na etapa de projeto foi definida a arquitetura do projeto através da elaboração de um diagrama de pacotes.

Na fase de implementação, foram desenvolvidas as funcionalidades corresponde as requisitos definidos. Por fim, na etapa de validação será feita a execução de um conjunto de experimentos: a seleção de programas na linguagem Java para serem utilizados durante a execução dos testes,a geração de conjuntos de casos de testes para os programas selecionados, avaliação do custo do tempo de execução e eficácia dos resultados da execução dos experimentos e análise comparativa entre a cobertura e eficiência(tempo) no conjunto de casos de teste reduzido e o conjunto de casos de teste original.

A Figura 5.2, apresenta o diagrama da etapas do projeto.

Figura 5.2 – Diagrama das etapas do projeto

Fonte – Elaborada pela Autora

5.4 ANÁLISE DE REQUISITOS

As Tabelas 5.2 e 5.3 apresentam os requisitos funcionais e não funcionais identificados no projeto AI+RTesting. O método de redução está representado pelos requisitos funcionais: RF01, RF02, RF03, RF10, RF11 e RF12.

Tabela 5.2 – Requisitos Funcionais.

Requisito	**Descrição**
RF01	Integrar a Ferramenta Mujava como um módulo da Jabuti.
RF02	Desenvolver rotinas para extrair e armazenar os dados da execução das Ferramentas Jabuti e Mujava.
RF03	Desenvolver rotina para escrever no arquivo Arff os resultados da execução das Ferramentas Jabuti e Mujava.
RF04	Integrar a ferramenta Weka a Jabuti através da biblioteca .jar.
RF05	Desenvolver rotinas para acionar os algoritmos de agrupamento.
RF06	Desenvolver rotina para utilizar o arquivo Arff gerado pela saída dos algoritmos de agrupamento como entrada do algoritmo J48.
RF07	Desenvolver rotinas para acionar o algoritmo de classificação J48.
RF08	Desenvolver rotinas para extrair e armazenar os resultados da aplicação do algoritmo de classificação J48.
RF09	Desenvolver rotinas para armazenar as regras de classificação geradas pelo J48 em uma estrutura de dados (árvore).
RF10	Desenvolver rotinas para redução casos de teste.
RF11	Desenvolver interface gráfica para redução de casos de teste.
RF12	Implementar a validação do conjunto reduzido indicando os percentuais de cobertura e o tempo gasto para execução do novo conjunto de teste.

Tabela 5.3 – Requisitos Não Funcionais.

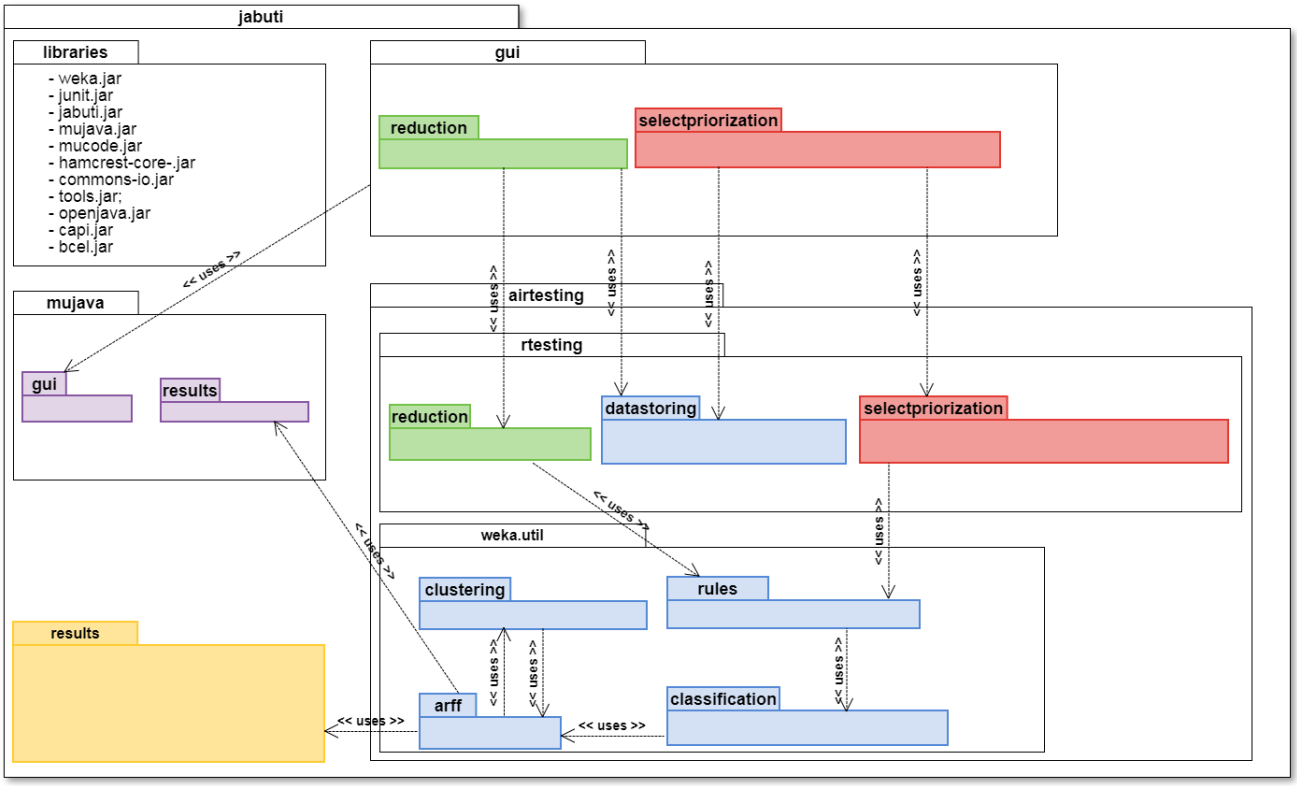
Requisito	**Descrição**
RNF01	Manter a modularidade das ferramentas.
RNF02	Documentar os códigos desenvolvidos para manutenção das ferramentas.
RNF03	Utilizar os padrões de desenvolvimento existentes na ferramenta Jabuti.

5.5 PROJETO E IMPLEMENTAÇÃO

Na etapa de análise de requisitos, foi elaborado o diagrama de pacotes do projeto AI+RTesting.Na Figura 5.3 são visualizados os pacotes que foram criados, adicionados e modificados dentro da ferramenta Jabuti.

Inicialmente, foi criado o pacote *results* para armazenar os resultados de execução dos testes das ferramentas Jabuti e Mujava. Em seguida, foi adicionado o pacote *weka.util* para possibilitar o acesso de todos os pacotes relacionados a aplicação das técnicas de aprendizado de máquina. Logo após, foi implementado o pacote *clustering* que contém as classes responsáveis pelas chamadas dos algoritmos de agrupamento. Além disso, foi criado o pacote *classification* para realizar as chamadas da aplicação do algoritmo de classificação e de escrita das regras geradas. Já o pacote *rules* armazena as classes desenvolvidas para interpretação das regras de classificação. Para organizar os métodos relacionados às três metodologias do teste de regressão foi criado o pacote *rtesting*. O pacote *gui* contém as interfaces necessárias para execução de todo o processo. Por último, as classes desenvolvidas para aplicação do método de redução estão armazenadas no pacote *reduction*.

Figura 5.3 – Diagrama de Pacotes AI+RTesting.



Fonte – Elaborada pela autora.

5.5.1 Ambiente de Desenvolvimento

A linguagem de programação escolhida para o desenvolvimento do projeto AI+Rtesting foi a linguagem Java. Devido ao fato de que as ferramentas Jabuti, Mujava e Weka que foram selecionadas para fazer parte do projeto são implementadas nesta linguagem de programação. Além disso, a proposta do AI+Rtesting é fornecer suporte para programas na linguagem java. Em vista disso, o desenvolvimento foi conduzido dentro do Ambiente de Desenvolvimento Integrado Eclipse que além de possuir uma orientação de desenvolvimento baseada em plug-ins, também oferece um amplo suporte para o desenvolvedor. Além disso, foi utilizado o Github para controle de versionamento.

5.5.2 Integração das Ferramentas Mujava e Jabuti

Com intuito de facilitar a execução do teste de regressão aplicando os métodos de redução, seleção e priorização de casos de teste, o projeto AI+RTesting propôs integração das ferramentas Jabuti e Mujava, para a execução do teste estrutural e do teste baseado em erros, respectivamente. Essa integração foi realizada em virtude da necessidade de que os dados da

execução dessas ferramentas fossem armazenados e integrados como se fossem apenas uma ferramenta.

Para isso, em primeiro lugar, a ferramenta Mujava foi acoplada a ferramenta Jabuti, se tornando um item de menu da mesma. Devido a isso, é possível executar os testes nas duas ferramentas fornecendo o mesmo conjunto de casos de teste. Em seguida, as informações das porcentagens de cobertura dos critérios da técnica estrutural e da técnica baseado em erros são extraídas e armazenadas em uma única estrutura de dados, que no caso é um *HashMap*.

Além disso, foi implementada a classe *CreateArtificialARFF* que é responsável pela geração automática de um arquivo no formato .arff com os dados que foram armazenado na estrutura de dados citada no parágrafo anterior. Para isso, nessa classe foram criados os atributos para cada critério estrutural e operador de mutação. Além disso, foram implementados os métodos para obter e setar o valor de cobertura referente a cada um desses atributos após a execução dos testes na Jabuti e Mujava. Uma amostra dessa classe pode ser visualizada no algoritmo 1.

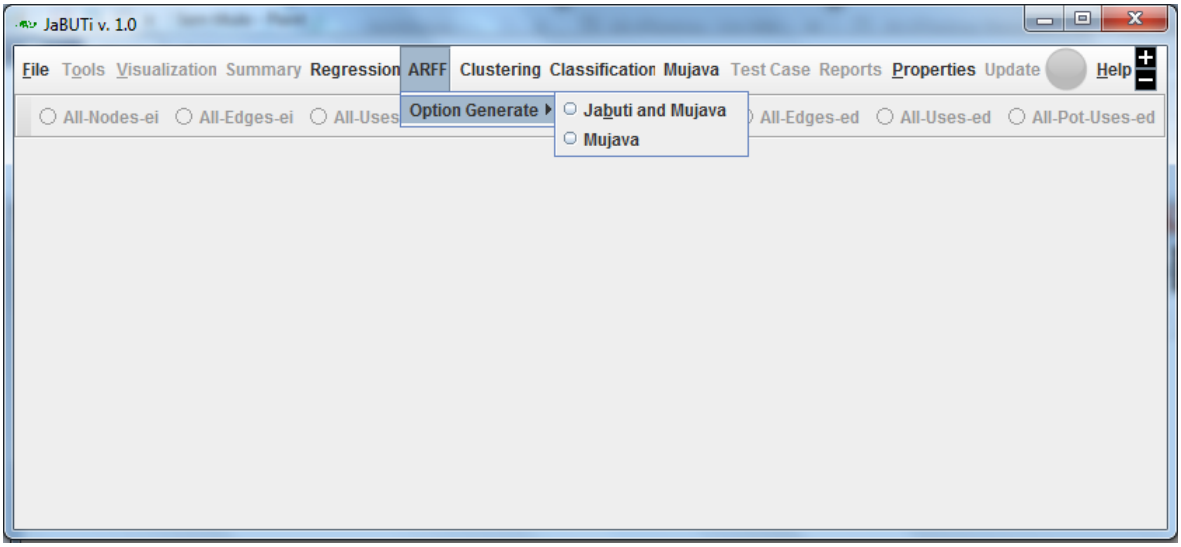
Algoritmo 1: Trecho da classe *CreateArtificialARFF*.

```
public void creatorRelation(String nomeRelacao){
  this.relation = new Instances(nomeRelacao,attributes,0);

  for (Entry<String,Map<String,Double>> kv :
     dataresultsJabutiMujava.entrySet()) {
    this.intanceData = new Instance(this.relation.
       numAttributes());
    this.intanceData.setValue(this.casesTest,kv.getKey());

    HashMap<String, Double> secondmap = (HashMap<String,
       Double>)kv.getValue();

    for(Entry<String,Double> kvv : secondmap.entrySet()){
     this.setValueIntanceData(kvv.getKey(),kvv.getValue());
    }
      this.relation.add(this.intanceData);
    }
    this.writerARFF(this.relation);
}
```

Essa classe é instanciada através de um item de menu na interface da ferramenta.

Quando instanciada pela primeira vez, a classe gera um .arff com os dados e a cada vez que os testes são executados novamente nas ferramentas, é possível sobrescrever o arquivo .arrf com o dados da nova execução. Para isso, é apenas necessário acionar a opção no menu mais uma vez. Esse arquivo é fornecido como entrada para a execução dos algoritmos de agrupamento, que por sua vez, também geram um arquivo de saída ARFF que posteriormente será usado como dados de entrada na execução do algoritmo de classificação.

Figura 5.4 – Geração do arff atráves da interface.



Fonte – Elaborada pela autora.

5.5.3 Implementação das Classes dos Algoritmos de Agrupamento

Os algoritmos de agrupamento disponibilizados pela API do Weka foram acionados através Das implementação das seguintes classes: *ClusterHierarchical*, *ClusterEM* e *ClusterSimpleKMeans*. Nessas classes foram implementados métodos para realizar as chamadas de cada um desses algoritmos e gerar os *clusters* associando aos casos de teste com os seus respectivos percentuais de cobertura dos critérios e operadores de mutação, como pode ser observado nas

figuras abaixo.

Algoritmo 2: Classe ClusterHierarchical

```
public void createClusterHierarchical(String dataTraining,
   String dataResult) {
    iData = removeAttributes.listIntances(iData);
    HierarchicalClusterer h = new HierarchicalClusterer();
    h.setNumClusters(10);
 }
```

Esse método é responsável por receber o parâmetro referente ao caminho do arquivo ARFF com o conjunto de dados que devem ser agrupados. Antes de realizar a operação de agrupamento, é necessário remover os atributos cujo valor é zero, assim apenas os percentuais válidos de cobertura de cada operador são levados em consideração no agrupamento. O número de classes é informado como o parâmetro do método *setNumClusters*.

Algoritmo 3: Classe Cluster EM

```
public void createClusterEM (String DataTraining, String
DataResult ) {
    FileReader reader = new FileReader(DataTraining);
    Instances iData = new Instances(reader);
    iData = removeAttributes.listIntances(iData);
    EM clusterer = new EM();
    clusterer.setNumClusters(4);
    clusterer.setSeed(18);
    clusterer.setMaxIterations(100);
    clusterer.setMinStdDev(1.0E-6);
 }
```

Assim como no método anterior, esse também recebe o parâmetro do caminho do arquivo ARFF com o conjunto de dados para e remove os atributos de valor zero. Além disso, os seguintes valores são passados como parâmetros de agrupamento: número de classes, o *seed* - responsável pela geração de números aleatórios, o número máximo de iterações que devem ser

executada e, por último, o valor mínimo para o desvio padrão ao calcular a densidade normal.

Algoritmo 4: Classe Cluster Simple K-Means

```
public void createClusterSimpleKMeans(String dataTraining,
   String DataResult){
    iData = removeAttributes.listIntances(iData);
    SimpleKMeans model = new SimpleKMeans();
    model.setNumClusters(4);
    model.setSeed(20);
    model.setMaxIterations(100);
    model.buildClusterer(iData);
}
```

Esse método realiza operações similares ao método citado anteriormente, com a exceção de que é instanciado um objeto do tipo *K-Means* e são passados como parâmetros de agrupamento apenas o número de classes, o *seed* e o número máximo de iterações para executar.

5.5.3.1 Implementação da Classe do Algoritmo de Classificação

O algoritmo de classificação J48 foi acionado atráves da implementação da classe *ClassifierJ48*. Essa classe tem a função de gerar as árvores de classificação a partir das classes obtidas pelos algoritmos de agrupamento.

Algoritmo 5: Método *callJ48* da classe *ClassifierJ48*

```
nada
public void callJ48(String path) throws Exception {
  FileReader reader = null;
  int index = returnValueIndex(path)-3;
  reader = new FileReader(path);
  Instances instances = null;
  instances = new Instances(reader);
  instances.setClassIndex(index);
  arvore.buildClassifier(instances);
}
```

Nessa classe foi implementado o método *callJ48*, ilustrado acima, que recebe como parâmetro o caminho do arquivo ARFF do conjunto de dados que será utilizado como a base de

dados de treinamento. Por intermédio desse método também é possível atribuir um índice de acordo com a ordem em que foi especificado no arquivo ARFF quando base for importada para a memória.

5.5.4 Método de Redução de Casos de Teste

Após a geração das regras fornecidas pelo algoritmo de classificação, foram analisadas as classes de equivalência para interpretar as características específicas de cada uma delas. Em seguida, foram identificadas quais eram as classes de teste mais importantes, que nesse contexto são as classes que cobrem determinado critério de teste ou geram um percentual maior de cobertura (LENZ, 2010b).

Para interpretar as regras de classificação, foi utilizada a seguinte estratégia: a estrutura de dados para armazenamento da regra foi implementada em formato de árvore, na qual o primeiro atributo da regra é o nó raiz. Posteriormente, foram identificados quais atributos seguintes devem ser inseridos à esquerda ou à direita do nó raiz da árvore, até que as folhas da árvore sejam os *clusters* associados à sua quantidade de casos de teste. No Algoritmo 6 pode-se visualizar um trecho da implementação para a interpretação das regras.

Algoritmo 6: Trecho do método de interpretação das regras de classficação

```
01: if(pesquisarElementoListaNos(quebrar[0]) == true){
02:     boolean nosExistem =
03:     pesquisarNosJaInserirdos(quebrar[0]);
04:       if(nosExistem == true){
05:            inserirCriteriosConcatenadosEsquerdo
06:            (quebrar[0], quebrar[3]);
07:            adicionarCluster(quebrar[3],
09             quebrar[4]);
08:            adicionaCriterioCluster(quebrar[3]);
09:         }
10:         else{
11:             inserirEsquerda(quebrar[0], quebrar[3]);
12:             adicionarCluster(quebrar[3], quebrar[4]);
13:             adicionaCriterioCluster(quebrar[3]);
```

Para determinar a quantidade de casos de testes que devem ser executados de cada classe, com a finalidade de reduzir o número de casos de teste e ainda assim garantir o percentual de cobertura dos critérios, foi utilizada a abordagem da Progressão Aritmética (PA) (CRISTO,

2017). Diferentemente da abordagem utilizada no trabalho de (SANTOS, 2018), no qual foi aplicada a Progressão Geométrica (PG).

Nessa abordagem, cada elemento da PA representa a porcentagem de casos de teste a ser selecionada para cada classe de equivalência. No primeiro momento, as classes são ordenadas de acordo com as regras geradas, logo após, a quantidade de casos de teste é selecionada com base na porcentagem gerada para o elemento da PA.

Para a classe de equivalência com menor prioridade é selecionado apenas um caso de teste. A partir do primeiro elemento da PA são gerados os demais elementos da PA, consequentemente, é obtida a quantidade de casos de teste a serem reexecutados durante o teste de regressão. Para tal, foram aplicados os seguintes passos:

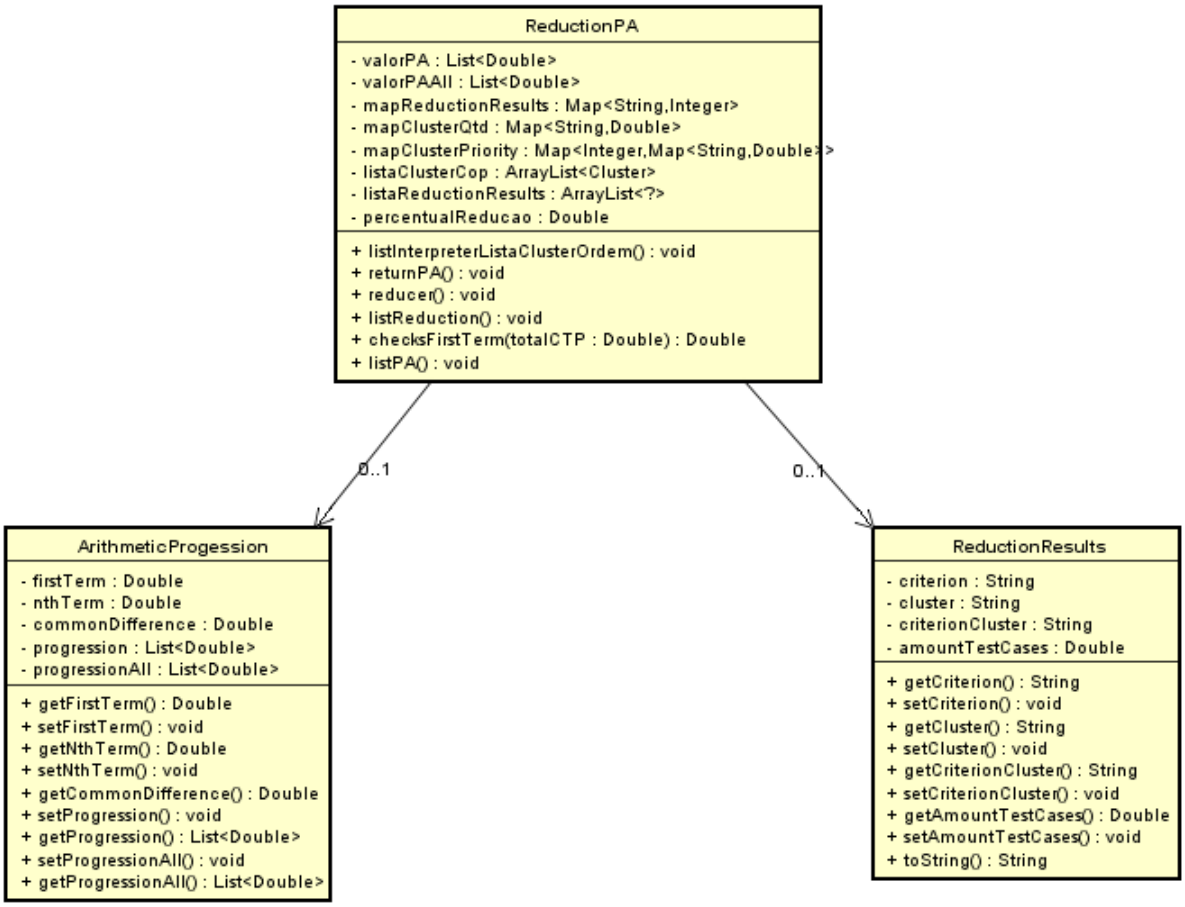
1. Define-se o valor do primeiro termo como 1.
2. Obtém-se o valor do último termo a partir do fator de redução informado pelo usuário.
3. Calcula-se a razão através das propriedades da PA.
4. A partir do valor da razão e da fórmula que calcula cada elemento da PA, obtém-se os valores dos outros elementos contidos na PA.

Para isso, foram implementadas as seguintes classes: *Reduction*, *ArithmeticProgression* e *ReductionResults*. O relaciomento entres essas classes ocorre da seguinte maneira:

1. A classe *Reduction* se relaciona com as classes *ArithmeticProgression* e *ReductionResults* atráves de uma associação unidirecional.
2. Na relação com a classe *ArithmeticProgression*, o valor de multiplicidade zero ou um.
3. Já no caso da relação *ReductionResults*, o valor de multiplicidade zero ou muitos.
4. Na classe *Reduction* foi implementado método *reducer()* que tem a responsabilidade de calcular o número de casos de teste que devem ser selecionados para redução por cada classe de equivalência definida pela regra gerada.
5. Esse cálculo é realizado utilizando a progressão gerada pelo método *setProgression(Double, Double, Double)* definido na classe *ArithmeticProgression*.
6. Em seguida, o resultado da redução é passado para a classe *ReductionResults*.

O relacionamento entre essas classe foi ilustrado no diagrama de classe da figura 5.5.

Figura 5.5 – Diagrama de Classe do Pacote *Reduction*



Fonte – Elaborada pela autora.

Na aplicação da PA foram utilizadas as seguintes fórmulas e propriedades abaixo:

Calcular o valor do elemento da PA:

$$a_n = a_1 + (n-1) * r \tag{5.1}$$

Obter a Soma dos temos da PA:

$$S_n = \frac{(a_1 + a_n) * n}{2} \tag{5.2}$$

De acordo com a propriedade da PA temos:

$$a_2 = a_1 + r \therefore a_3 = a_2 + r = a_1 + 2r \therefore a_4 = a_3 + r = a_1 + 3r \cdots \tag{5.3}$$

A soma dos elementos equidistantes de uma PA possuem valores iguais. Assim, podemos obter o valor da razão. Supondo que o valor do primeiro termo da PA seja 1 e o do

último termo seja 19, calculamos a razão:

$$a_1 + a_n = valor \therefore 1 + 19 = 20 \tag{5.4}$$

$$a_1 + a_n = valor \therefore 1 + 1 + 3r = 20 \therefore r = \frac{20-2}{3} = 6 \tag{5.5}$$

5.6 CONSIDERAÇÕES FINAIS

Este capítulo abordou toda a metodologia empregada para a realização do trabalho, desde a análise de requisitos até o detalhamento do método proposto pelo presente trabalho, enfatizando a descrição do ambiente de desenvolvimento, a integração das ferramentas Mujava e Jabuti e a aplicação da Progressão Aritmética para auxiliar na identificação dos casos de teste que devem ser selecionados no método de redução. No próximo capitulo, será relatado todo o processo de execução dos experimentos e a validação do método proposto.

6 EXPERIMENTOS E RESULTADOS

Neste capítulo, é abordado de forma detalhada todo processo de condução, execução e validação dos experimentos. Além disso, apresenta a análise dos resultados obtidos pelos novos conjuntos de casos de testes definidos pelo método proposto em comparação aos resultados do conjunto mínimo.

6.1 OBJETIVO DOS EXPERIMENTOS

Com o objetivo de validar a solução proposta foi definido um roteiro de experimentos que é constituído dos seguintes passos:

1. Selecionar métodos de teste e criar conjuntos de casos de teste mínimos e redundantes para os métodos selecionados
2. Executar o teste baseado em erros para o obter os percentuais de cobertura dos operadores de mutação cada caso de teste do conjunto original
3. Executar o método de redução com os fatores de redução 50, 80 e 100 para auxiliar na geração do novo conjunto de teste
4. Executar o teste estrutural e o teste baseado em erros com o novo conjunto de testes
5. Comparar o percentual de cobertura dos critérios estruturais e escore de mutação do novo conjunto de testes com o percentual do conjunto original

6.2 CONDUÇÃO DOS EXPERIMENTOS

Para a execução dos experimentos foram implementados dois algoritmos: o método *MDC* (Máximo Divisor Comum) e o método *Pirâmide*. No MDC são informados dois números inteiros como entrada e é calculado o máximo divisor comum entre eles, como ilustrado no algoritmo 7. No *Pirâmide*, por sua vez, são fornecidos seis valores inteiros de entrada e é

determinado se é possível formar uma pirâmide triangular regular.

Algoritmo 7: Método *MDC*.

```
public class MDC {
      public static int calcularMDC(int a, int b){
        if(a == b && b == 0){ //12
          return -1;
        }
        if (a < -2147483640 || a > 2147483640)
                return -1;
        if (b < -2147483640 || b > 2147483640)
                return -1;
        if(a < 0)
                a = -a;
        if(b < 0){
            b = -b;
            int r = 0;
            while (b != 0)
              r = a % b;
        }
}
```

6.2.1 Geração de Casos de Teste

Visando validar os métodos citados acima, em primeiro lugar, foram implementadas as classes de teste que contém o conjunto mínimo de casos de teste necessários para alcançar maior cobertura possível. Para o método MDC, a classe de teste *MDCTest* é composta por 26 casos de teste. Já no método Pirâmide, a classe de teste *PiramideTest* compreende 30 casos de teste.

Em seguida, foram criadas as classes de teste redundantes, ou seja, para cada caso de teste do conjunto mínimo foram implementados entre 2 e 3 casos de teste similar. As classes *MDCRedundanteTest* e *PiramideRedundanteTest* possuem 101 e 104 casos de testes redundantes, respectivamente.

Esses casos foram propositalmente acrescidos no conjunto de caso de teste original para que posteriormente sejam utilizados para validar se o método proposto é capaz de agrupar, classificar e remover essas redundâncias do conjunto. No algoritmo 8, é possível observar alguns dos casos de teste do *MDCRedundanteTest*.

Algoritmo 8: Classe *MDCRedundanteTest*

```
public class MDCRedundanteTest {
    @Test
  public void mdc_0_0() {
    int a = 0;
    int b = 0;
    int valorEsperado = -1;
    int valorAlcancado = MDC.calcularMDC(a, b);
    assertEquals(valorEsperado, valorAlcancado);
  }
  @Test
  public void mdcParametroAZeroParametroBZero01() {
    int a = 0;
    int b = 0;
    int valorEsperado = -1;
    int valorAlcancado = MDC.calcularMDC(a, b);
    assertEquals(valorEsperado, valorAlcancado);
  }
}
```

Após a geração das classes de casos de teste redundantes, as mesmas foram submetidos as ferramentas Jabuti e Mujava para que fossem realizados o teste estrutural e baseado em erros e coletados os valores referentes a cobertura dos critérios estruturais e escore de mutação.

Tabela 6.1 – Resultados da execução do teste estrutural e baseado em erros para o método *MDC*

Todos-Nós	**Todos-Arcos**	**Todos-Usos**	**Todos-PUsos**	**Score Mutação**	**Tempo**
98%	98%	98%	98%	92%	00:00:00

Tabela 6.2 – Resultados da execução do teste estrutural e baseado em erros para o método *Pirâmide*

Todos-Nós	**Todos-Arcos**	**Todos-Usos**	**Todos-PUsos**	**Score Mutação**	**Tempo**
100%	100%	96%	97%	93%	00:00:00

Os casos de teste foram classificados conforme as suas respectivas classes de equi-

valência.Nas Tabelas 6.3 e 6.4 foram indicadas as classes de equivalência e sua respectiva quantidade de casos de teste do método *MDC* e *Pirâmide*.

Tabela 6.3 – Classes de equivalência do método *MDC*.

Classe Equivalência	**Descrição**	**Casos de Teste**
0	A igual a B	1
1	A zero e B zero	1
2	A zero e B negativo	3
3	A negativo e B zero	7
4	A zero e B positivo	3
5	A positivo e B zero	3
6	A negativo e B negativo	2
7	A negativo e B positivo	3
8	A positivo e B negativo	1
9	A positivo e B positivo	2
Total		26

Tabela 6.4 – Classes de equivalência do método *Pirâmide*.

Classe Equivalência	**Descrição**	**Casos de Teste**
0	Variáveis fora dos limites máximo	6
1	Variáveis fora dos limites mínimo	6
2	Valores da base e faces em ordem crescente	7
3	Valores da base e faces não formam triângulos	2
4	Valores da base e faces não formam triângulos equiláteros	5
5	Variáveis formam uma pirâmide triangular regular	4
Total		30

6.3 EXECUÇÃO DOS TESTES

Com o intuito de coletar os percentuais de cobertura dos critérios estruturais, inicialmente, as classes de teste foram executadas no JUnit e, em seguida, as mesmas foram submetidas às ferramentas Jabuti e Mujava. As Tabelas 6.5 e 6.7 apresentam uma amostra desses dados percentuais de cobertura do teste estrutural para os casos de teste do método *MDC* e *Pirâmide*. Já as Tabelas 6.6 e 6.8 apresentam os dados percentuais de cobertura dos operadores de mutação.

Tabela 6.5 – Amostra de cobertura do teste estrutural do método *MDC*.

Casos de Teste	**Nós**	**Arcos**	**Usos**	**Potenciais Usos**
Máximo divisor comum	57%	47%	42%	39%
A valor positivo	21%	13%	12%	10%
A valor negativo	15%	8%	9%	7%
A e B valor negativo	73%	60%	48%	43%
A e B valor positivo	31%	21%	18%	15%

Tabela 6.6 – Amostra de cobertura do teste baseado em erros do método *MDC*.

Casos de Teste	**AOIS**	**AODU**	**AOIU**	**AORB**	**VDL**
Máximo divisor comum	79%	0%	100%	100%	45%
A valor positivo	15%	20%	0%	50%	100%
A valor negativo	12%	20%	0%	50%	100%
A e B valor negativo	86%	20%	50%	75%	100%

Tabela 6.7 – Amostra de cobertura do teste estrutural do método *Pirâmide*.

Casos de Teste	**Nós**	**Arcos**	**Usos**	**Potenciais Usos**
Base maior que 15	10%	3%	6%	4%
Base menor que 15	7%	2%	4%	3%
Base crescente 1	38%	15%	23%	22%
Base não forma triângulo 1	46%	23%	27%	26%

Tabela 6.8 – Amostra de cobertura do teste baseado em erros do método *Pirâmide*.

Casos de Teste	**AOIS**	**AODU**	**AOIU**	**AORB**	**VDL**
Base maior que 1	1%	11%	100%	0%	0%
Base menor que 15	0%	11%	100%	0%	0%
Base crescente 1	5%	11%	100%	0%	0%
Base não forma triângulo 1	8%	11%	100%	12%	0%

Os dados coletados foram armazenados em um arquivo Arff que contém os seguintes atributos: o nome dos casos de testes, o nome dos critérios estruturais e os percentuais de cobertura. Na figura 6.1 foi ilustrado uma amostra do arquivo gerado pela ferramenta.

Figura 6.1 – Amostra do arquivo Arff gerado pela ferramenta

```
@relation AI+RTesting

@attribute 'Casos de teste' {}
@attribute AOIS numeric
@attribute AODU numeric
@attribute AOIU numeric
@attribute AORB numeric
@attribute VDL numeric
@attribute SDL numeric
@attribute COI numeric
@attribute COR numeric
@attribute LOI numeric
@attribute ROR numeric
@attribute AORS numeric
@attribute AODS numeric
@attribute COD numeric
@attribute SOR numeric
@attribute LOD numeric
@attribute CDL numeric
@attribute LOR numeric
@attribute ASRS numeric
@attribute ODL numeric
@attribute NODES numeric
@attribute EDGES numeric
@attribute USES numeric
@attribute PUSES numeric

@data
face2NaoFormaTrianguloEquilatero_10,85,11,100,75,100,51,100,38,60,57,0,0,0,0,0,0,0,0,10,21,42,25,43
face1NaoFormaTrianguloEquilatero1_11,41,11,100,75,100,41,90,35,48,48,0,0,0,0,0,0,0,0,10,19,37,19,38
face1NaoFormaTrianguloEquilatero2_12,52,11,100,75,100,41,93,41,48,50,0,0,0,0,0,0,0,0,12,20,39,41,40
baseNaoFormaTrianguloEquilatero2_07,5,11,100,37,50,27,72,38,27,37,0,0,0,0,0,0,0,0,8,15,29,27,30
face1NaoFormaTrianguloEquilatero1_08,41,11,100,75,100,41,90,35,48,48,0,0,0,0,0,0,0,0,10,19,37,39,38
baseNaoFormaTrianguloEquilatero1_06,0,11,100,37,50,27,69,32,27,35,0,0,0,0,0,0,0,0,6,15,28,25,29
face2NaoFormaTrianguloEquilatero_05,85,11,100,75,100,51,100,38,60,57,0,0,0,0,0,0,0,0,10,21,42,45,43
```

Fonte – Elaborada pela autora.

6.3.1 Parâmetros de Configuração dos Algoritmos de Agrupamento

Os parâmetros de configuração utilizados para executar os algoritmos de agrupamento nesse trabalho foram baseados nos parâmetros aplicados nos trabalhos de Santos (2016), Cristo (2017) e Santos (2018), pois os mesmos alcançaram resultados satisfatórios. Esses parâmetros foram ilustrados na tabela 6.9.

Tabela 6.9 – Parâmetros dos Experimentos dos Algoritmos de Agrupamento.

Experimento	***clusters***	**MaxIterations**	**MinStdDev**	**Seed**
1.1	10	100	1.0E-6	18
1.2	10	100		20
1.3	10			
2.1	6	100	1.0E-6	20
2.2	6	100		20
2.3	6			

Em cada experimento foi considerado a quantidade de classes de equivalência identificadas manualmente e que geraram casos de teste para estabelecer o parâmetro do número de *clusters* na execução do agrupamento.

A fim de facilitar o entendimento dos experimentos, os mesmos foram numerados. Para os métodos de teste temos os seguintes valores: 1. Pirâmide e 2. MDC. Já os algoritmos de agrupamento foram valorados como: 1. EM, 2. K-means e 3. Agrupamento Hierárquico.

6.4 IDENTIFICAÇÃO DAS CLASSES DE EQUIVALÊNCIA

Após a obtenção dos resultados dos experimentos do teste estrutural e baseado em erros, foram executados os algoritmos de agrupamento para separar os casos de teste em classes de equivalência(*clusters*). As tabelas 6.10, 6.11, 6.12, 6.13, 6.14 e 6.15 apresentam o número de *clusters* e a quantidade de casos de teste em cada um deles.

Tabela 6.10 – Classes de equivalências geradas pelos experimentos 1.1.

Classe de Equivalência	**Casos de Teste**
0	4
1	16
2	20
3	4
4	4
5	4
6	12
7	8
8	16
9	16
Total	104

Tabela 6.11 – Classes de equivalências geradas pelos experimentos 1.2.

Classe de Equivalência	**Casos de Teste**
0	16
1	12
2	16
3	8
4	3
5	8
6	16
7	1
8	4
9	20
Total	104

Tabela 6.12 – Classes de equivalências geradas pelos experimentos 1.3.

Classe de Equivalência	**Casos de Teste**
0	24
1	16
2	4
3	16
4	4
5	12
6	4
7	4
8	4
9	16
Total	104

Tabela 6.13 – Classes de equivalências geradas pelo experimento 2.1.

Classe de Equivalência	**Casos de Teste**
0	13
1	24
2	14
3	20
4	15
5	16
Total	102

Tabela 6.14 – Classes de equivalências geradas pelo experimento 2.2.

Classe de Equivalência	**Casos de Teste**
0	16
1	14
2	4
3	40
4	17
5	11
Total	102

Tabela 6.15 – Classes de equivalências geradas pelo experimento 2.3.

Classe de Equivalência	**Casos de Teste**
0	14
1	30
2	11
3	3
4	40
5	4
Total	102

6.5 CLASSIFICAÇÃO

Depois de definidas as classes de equivalências e seus respectivos casos de teste, foi possível executar o algoritmo de classificação J48 para gerar as regras necessárias para realizar os experimentos de redução no conjunto de casos de teste redundantes. Essas regras são representadas em forma de árvore de decisão, onde cada nó da árvore representar uma decisão a ser tomada a partir dos dados de entrada. Deste modo foi estabelecida a ordem de prioridade dos casos de testes com base no valor percentual da cobertura de cada critério. Nas Figuras 6.2, 6.3, 6.4, 6.5, 6.6 e 6.7 estão ilustradas as regras de classificação geradas pelos experimentos com o J48.

Figura 6.2 – Regra gerada para o experimento 1.1.

```
COR <= 16
|   COR <= 0
|   |   AODU <= 0
|   |   |   NOS <= 26: cluster1 (16.0)
|   |   |   NOS > 26: cluster5 (4.0)
|   |   AODU > 0
|   |   |   NOS <= 31: cluster6 (12.0)
|   |   |   NOS > 31: cluster0 (4.0)
|   COR > 0
|   |   NOS <= 0: cluster3 (4.0)
|   |   NOS > 0: cluster2 (20.0)
COR > 16
|   NOS <= 26
|   |   AODU <= 0: cluster9 (16.0)
|   |   AODU > 0: cluster8 (16.0)
|   NOS > 26
|   |   AOIU <= 25: cluster7 (8.0)
|   |   AOIU > 25: cluster4 (4.0)

Number of Leaves  :    10

Size of the tree :     19
```

Fonte – Elaborada pela autora.

Para o experimento 1.1 que foi utilizado o algoritmo EM e o método *MDC* a figura 6.2 demostra que a regra foi definida com base em quatro atributos: os operadores de mutação COR (substitui operadores condicionais binários por outros operadores condicionais), AOIU (insere um operador aritmético unário básico), AODU (exclui operadores aritméticos unários básicos) e o critério NOS(Todos-Nós), sendo que o atributo de maior prioridade foi o COR.

Figura 6.3 – Regra gerada para o experimento 1.2.

```
COR <= 16
|   COR <= 0
|   |   NOS <= 71: cluster2 (16.0)
|   |   NOS > 71: cluster9 (20.0)
|   COR > 0
|   |   NOS <= 26
|   |   |   COI <= 16
|   |   |   |   NOS <= 0: cluster5 (4.0)
|   |   |   |   NOS > 0: cluster1 (12.0)
|   |   |   COI > 16: cluster5 (4.0)
|   |   NOS > 26: cluster8 (4.0)
COR > 16
|   AODU <= 0
|   |   AOIU <= 25: cluster0 (16.0)
|   |   AOIU > 25: cluster4 (4.0/1.0)
|   AODU > 0
|   |   NOS <= 26: cluster6 (16.0)
|   |   NOS > 26: cluster3 (8.0)

Number of Leaves  :    10

Size of the tree :     19
```

Fonte – Elaborada pela autora.

A figura 6.3 demostra que a regra gerada para o experimento 1.2 que utilizou o algoritmo K-means e o método *MDC*, assim como no algoritmo anterior, foi definida de acordo com cinco atributos: os operadores de mutação COR (substitui operadores condicionais binários por outros operadores condicionais), AOIU (insere um operador aritmético unário básico), AODU (exclui operadores aritméticos unários básicos) e o critério NOS(Todos-Nós). Nessa caso, o operador de mutação COR teve a maior prioridade.

Figura 6.4 – Regra gerada para o experimento 1.3.

```
AODU <= 0
|   COR <= 16
|   |   NOS <= 26: cluster9 (16.0)
|   |   NOS > 26: cluster6 (4.0)
|   COR > 16
|   |   NOS <= 26: cluster1 (16.0)
|   |   NOS > 26: cluster2 (4.0)
AODU > 0
|   COR <= 16
|   |   COR <= 0
|   |   |   NOS <= 31: cluster5 (12.0)
|   |   |   NOS > 31: cluster4 (4.0)
|   |   COR > 0: cluster0 (24.0)
|   COR > 16
|   |   NOS <= 26: cluster3 (16.0)
|   |   NOS > 26
|   |   |   AOIS <= 36: cluster7 (4.0)
|   |   |   AOIS > 36: cluster8 (4.0)

Number of Leaves  :    10

Size of the tree :    19
```

Fonte – Elaborada pela autora.

De acordo com a figura 6.4 o experimento 1.3 que utilizou o algoritmo de Agrupamento Hierárquico e o método *MDC*, estabeleceu a regra em três atributos: os operadores de mutação AODU (exclui operadores aritméticos unários básicos), COR (substitui operadores condicionais binários por outros operadores condicionais) e o critério NOS(Todos-Nós). Para esse algoritmo o atributo de maior prioridade foi o AODU.

Figura 6.5 – Regra gerada para experimento 2.1.

```
VDL <= 0
|   ROR <= 2
|   |   COI <= 9: cluster5 (16.0)
|   |   COI > 9: cluster1 (24.0)
|   ROR > 2: cluster0 (13.0)
VDL > 0
|   AODU <= 0: cluster2 (14.0)
|   AODU > 0
|   |   VDL <= 50: cluster3 (20.0)
|   |   VDL > 50: cluster4 (15.0)

Number of Leaves  :    6

Size of the tree :    11
```

Fonte – Elaborada pela autora.

A figura 6.5 demonstra que para o experimento 2.1, no qual foi aplicado o algoritmo EM e o método Pirâmide, a regra foi definida utilizando os operadores de mutação: VDL (exclui todas as referências de variáveis de todas as expressões), ROR (substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso), COI (insere operadores condicionais unários) e AODU(exclui operadores aritméticos unários básicos), sendo o primeiro como o operador de maior prioridade.

Figura 6.6 – Regra gerada para o experimento 2.2.

```
ROR <= 2: cluster3 (40.0)
ROR > 2
|   VDL <= 50
|   |   ROR <= 35: cluster4 (16.0)
|   |   ROR > 35: cluster0 (17.0/1.0)
|   VDL > 50
|   |   AODU <= 0: cluster1 (14.0)
|   |   AODU > 0
|   |   |   COR <= 35: cluster2 (4.0)
|   |   |   COR > 35: cluster5 (11.0)

Number of Leaves  :    6

Size of the tree :    11
```

Fonte – Elaborada pela autora.

Para o experimento 2.2, ilustrado na figura acima, onde foi aplicado o algoritmo K-means e o método Pirâmide, os atributos foram baseados nos operadores de mutação: ROR (substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso), VDL (exclui todas as referências de variáveis de todas as expressões), AODU(exclui operadores aritméticos unários básicos) e COR (substitui operadores condicionais binários por outros operadores condicionais). Nesse experimento, o ROR foi o mais prioritário.

Figura 6.7 – Regra gerada para o experimento 2.3.

```
ROR <= 2: cluster4 (40.0)
ROR > 2
|   AODU <= 0
|   |   AORB <= 75: cluster2 (11.0)
|   |   AORB > 75: cluster3 (3.0)
|   AODU > 0
|   |   NOS <= 61
|   |   |  NOS  <= 28: cluster5 (4.0)
|   |   |  NOS  > 28: cluster1 (30.0)
|   |   NOS  > 61: cluster0 (14.0)

Number of Leaves  :    6

Size of the tree :     11
```

Fonte – Elaborada pela autora.

Já no experimento 2.3, no qual foi utilizado o algoritmo hierárquico, os operadores de mutação utilizados foram ROR (substitui operadores relacionais por outros operadores relacionais e substitui todo o predicado com verdadeiro e falso), AODU(exclui operadores aritméticos unários básicos), AORB (substitui operadores aritméticos binários por outros operadores aritméticos binários) e o critério NOS(Todos-Nós). Assim como na regra gerada pelo algoritmo citado anteriormente, o operador de mutação de maior prioridade foi ROR, como é possível observar na figura 6.7.

6.6 REDUÇÃO

Após a geração das regras de classificação, foi possível identificar as características das classe de equivalência e estabelecer a ordem de prioridade de cada uma delas. Essas informações são imprescindíveis para conduzir o testador no momento da seleção de casos de

testes que devem ser executados para que seja obtido um alto percentual de cobertura com um menor número de casos possível (LENZ, 2010b).

No trabalho de Cristo Cristo (2017) foi proposto a utilização de Progressão Aritmética (PA) para determinar de maneira sistemática o número de casos de testes que devem ser executados em cada classe de equivalência, sendo que este corresponderia cada elemento da PA (CRISTO, 2017). Tendo como base esta abordagem, primeiramente, são ordenadas as classes, respeitando a ordem de prioridade definida pelas regras do J48. Em seguida, é selecionado o número de casos de teste considerando a porcentagem de cada elemento da PA.

Nas classes de equivalência de menor prioridade, é selecionado apenas um caso de teste, sendo que este é atribuído como o primeiro elemento da PA. A partir disso, é calculada a razão da PA e os demais elementos da PA. Assim obtendo a quantidade de testes que devem ser executados em cada classe. Esses dados serviram de auxílio para testador na realização do teste de regressão.

6.6.1 Aplicação das Regras

Os experimentos de redução foram realizados de acordo com as regras estabelecidas na seção 5.4.4. Além dessas regras, no processo de execução da redução, o testador também deve indicar o Fator de Redução (FR) desejado e, em seguida, esse valor é atribuído como enésimo termo da PA. No presente trabalho, com o objetivo de analisar a tendência de cobertura dos critérios do Teste Estrutural, foram utilizados os seguintes FR: 50, 80 e 100. Estes fatores foram aplicados no trabalho de Santos (2018) e os mesmos obtiveram resultados aceitáveis. Após a realização dos experimentos de redução, foram gerados novos conjuntos de casos de teste com base nas informações obtidas. Para facilitar a compreensão dos experimentos, foram numerados os FR da seguinte forma: 1. 50, 2. 80 e 3. 100.

6.6.1.1 Aplicação das Regras no MDC

Primeiramente, os experimentos com o método pirâmide foram feitos com o algoritmo EM para cada um dos fatores de redução mencionados na seção anterior. O interpretador da árvore definiu que o atributo de maior prioridade foi o COR, sendo que este situa-se na raiz da árvore, dessa forma foi estabelecida a ordem de prioridade dos *clusters*. Os resultados podem ser observados nas tabelas 6.16, 6.17 e 6.18.

Tabela 6.16 – Número de casos de teste selecionados para o experimento 1.1.1

Classe de Equivalência	Valores PA	Casos de Teste
8	0.5	6
9	0.396	4
2	0.314	4
7	0.249	4
4	0.198	2
6	0.157	2
1	0.124	1
0	0.099	1
5	0.078	1
3	0.062	1
Total		26

Tabela 6.17 – Número de casos de teste selecionados para o experimento 1.1.2

Classe de Equivalência	Valores PA	Casos de Teste
8	0.603	8
9	0.454	6
2	0.342	6
7	0.257	5
4	0.193	4
6	0.146	2
1	0.11	1
0	0.082	1
5	0.062	1
3	0.042	1
Total		35

Tabela 6.18 – Número de casos de teste selecionados para o experimento 1.1.3

Classe de Equivalência	Valores PA	Casos de Teste
8	1.0	9
9	0.734	7
2	0.539	6
7	0.396	6
4	0.291	4
6	0.213	2
1	0.157	1
0	0.115	1
5	0.084	1
3	0.062	1
Total		38

Posteriormente, os experimentos foram feitos com o algoritmo K-means para todos

os FR. Para esse algoritmo, o interpretador da árvore definiu que o atributo de maior prioridade foi o COR, sendo que está situa-se no raiz da árvore, dessa forma foi estabelecida a ordem de prioridade dos *clusters*, como demonstrado nas tabelas 6.19, 6.20 e 6.21.

Tabela 6.19 – Número de casos de teste selecionados para o experimento 1.2.1

Classe de Equivalência	**Valores PA**	**Casos de Teste**
6	0.396	7
3	0.314	4
0	0.249	4
1	0.198	2
9	0.157	2
4	0.124	2
8	0.099	1
5	0.078	1
2	0.062	1
Total		24

Tabela 6.20 – Número de casos de teste selecionados para o experimento 1.2.2

Classe de Equivalência	**Valores PA**	**Casos de Teste**
6	0.603	10
3	0.454	7
0	0.342	6
1	0.257	2
9	0.193	2
4	0.124	2
8	0.11	2
5	0.082	1
2	0.062	1
Total		33

Tabela 6.21 – Número de casos de teste selecionados para o experimento 1.2.3

Classe de Equivalência	**Valores PA**	**Casos de Teste**
6	0.734	12
3	0.539	8
0	0.396	7
1	0.291	2
9	0.213	2
4	0.157	3
8	0.115	2
5	0.084	1
2	0.062	1
Total		38

Por último, foi utilizado o algoritmo hierárquico para executar os experimentos de redução com o Pirâmide. Neste caso, o atributo de maior prioridade foi o AODU. Os resultados obtidos estão apresentados nas tabelas 6.22, 6.23 e 6.24.

Tabela 6.22 – Número de casos de teste selecionados para o experimento 1.3.1

Classe de Equivalência	**Valores PA**	**Casos de Teste**
0	0.5	6
3	0.396	6
8	0.314	2
7	0.249	2
5	0.198	2
1	0.157	2
4	0.124	1
6	0.099	1
9	0.078	1
2	0.062	1
Total		24

Tabela 6.23 – Número de casos de teste selecionados para o experimento 1.3.2

Classe de Equivalência	**Valores PA**	**Casos de Teste**
0	0.5	6
3	0.396	6
8	0.314	2
7	0.249	2
5	0.198	2
1	0.157	2
9	0.124	1
6	0.099	1
4	0.078	1
2	0.062	1
Total		34

Tabela 6.24 – Número de casos de teste selecionados para o experimento 1.3.3

Classe de Equivalência	Valores PA	Casos de Teste
0	1.0	9
3	0.734	8
8	0.539	4
7	0.396	3
5	0.291	3
1	0.213	2
4	0.157	1
6	0.115	2
9	0.084	1
2	0.062	1
Total		36

6.6.1.2 Aplicação das Regras no Pirâmide

Os experimentos com o método pirâmide foram feito com o algoritmo EM para cada um dos fatores de redução. O interpretador da árvore definiu que o atributo de maior prioridade foi o VDL que está localizado na raiz da árvore, assim estabelecendo a ordem de prioridade dos *clusters*. Os resultados podem ser observados nas tabelas 6.25, 6.26 e 6.27.

Tabela 6.25 – Número de casos de teste selecionados para o experimento 2.1.1

Classe de Equivalência	Valores PA	Casos de Teste
4	0.5	8
3	0.329	7
2	0.217	4
1	0.143	3
0	0.094	2
5	0.062	1
Total		25

Tabela 6.26 – Número de casos de teste selecionados para o experimento 2.1.2

Classe de Equivalência	Valores PA	Casos de Teste
4	0.52	12
3	0.48	10
2	0.288	5
1	0.172	3
0	0.103	3
5	0.062	1
Total		34

Tabela 6.27 – Número de casos de teste selecionados para o experimento 2.1.3

Classe de Equivalência	Valores PA	Casos de Teste
4	0.62	15
3	0.574	12
2	0.329	5
1	0.189	3
0	0.108	3
5	0.062	1
Total		39

Posteriormente, os experimentos foram feitos com o algoritmo K-means para todos os FR. Neste algoritmo, o interpretador da árvore definiu que o atributo de maior prioridade foi o ROR que encontra-se no raiz da árvore, desse modo foi estabelecida a ordem de prioridade dos *clusters*, como demonstrado nas tabelas 6.28, 6.29 e 6.30.

Tabela 6.28 – Número de casos de teste selecionados para o experimento 2.2.1

Classe de Equivalência	Valores PA	Casos de Teste
5	0.32	6
1	0.275	3
0	0.151	2
2	0.083	2
4	0.046	1
3	0.025	1
Total		15

Tabela 6.29 – Número de casos de teste selecionados para o experimento 2.2.2

Classe de Equivalência	Valores PA	Casos de Teste
5	0.5	9
1	0.4	3
0	0.2	2
2	0.1	2
4	0.05	1
3	0.025	1
Total		18

Tabela 6.30 – Número de casos de teste selecionados para o experimento 2.2.3

Classe de Equivalência	Valores PA	Casos de Teste
5	0.999	11
1	0.478	4
0	0.229	2
2	0.109	2
4	0.052	1
3	0.025	1
Total		21

Por último, foi utilizado o algoritmo hierárquico para executar os experimentos de redução com o Pirâmide. Do mesmo modo que o algoritmo citado anteriormente, o atributo de maior prioridade foi o ROR. Os resultados obtidos estão apresentados nas tabelas 6.31, 6.32 e 6.33.

Tabela 6.31 – Número de casos de teste selecionados para o experimento 2.3.1

Classe de Equivalência	Valores PA	Casos de Teste
1	0.32	9
0	0.275	7
5	0.151	1
4	0.083	1
3	0.046	1
2	0.025	1
Total		20

Tabela 6.32 – Número de casos de teste selecionados para o experimento 2.3.2

Classe de Equivalência	Valores PA	Casos de Teste
1	0.8	12
0	0.4	12
5	0.2	1
4	0.1	1
3	0.05	1
2	0.025	1
Total		28

Tabela 6.33 – Número de casos de teste selecionados para o experimento 2.3.3

Classe de Equivalência	Valores PA	Casos de Teste
1	0.999	15
0	0.478	14
5	0.229	1
4	0.109	1
3	0.052	1
2	0.025	1
Total		33

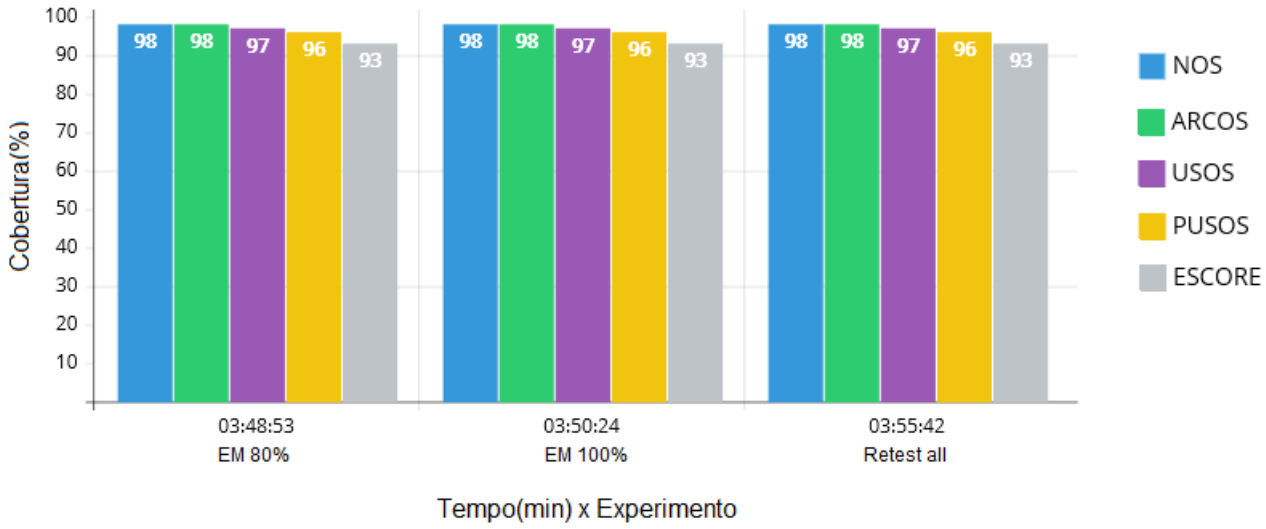
6.7 ANÁLISE DE RESULTADOS

Após a realização dos experimentos dos método de redução foi definido a quantidade de casos de teste de cada classe de equivalência. Com isso, foi possível gerar os novos conjuntos de teste para coletar o tempo de execução e o percentual de cobertura, e posteriormente, comparar com as informações da abordagem do *retest all*. Assim, validando a eficiência do método proposto. Os resultados comparativos da quantidade de casos de teste e o percentual de cobertura da execução dos algoritmos de teste *MDC* e *Pirâmide* com relação a abordagem retest all foram apresentados atráves de gráficos.

6.7.1 Análise de Resultados para o Método *MDC*

Na Figura 6.8, é possível visualizar as porcentagens de cobertura dos critérios de teste e o tempo de execução alcançados pelos experimentos de redução com o algoritmo EM e o método *MDC*. Para os fatores de redução 80 e 100, o algoritmo EM alcançou as porcentagens de cobertura dos critérios estruturais da abordagem retest all e reduziu o tempo de execução. Já com o fator de redução de 50%, não foi alcançado o percentual de cobertura do *retest all*.

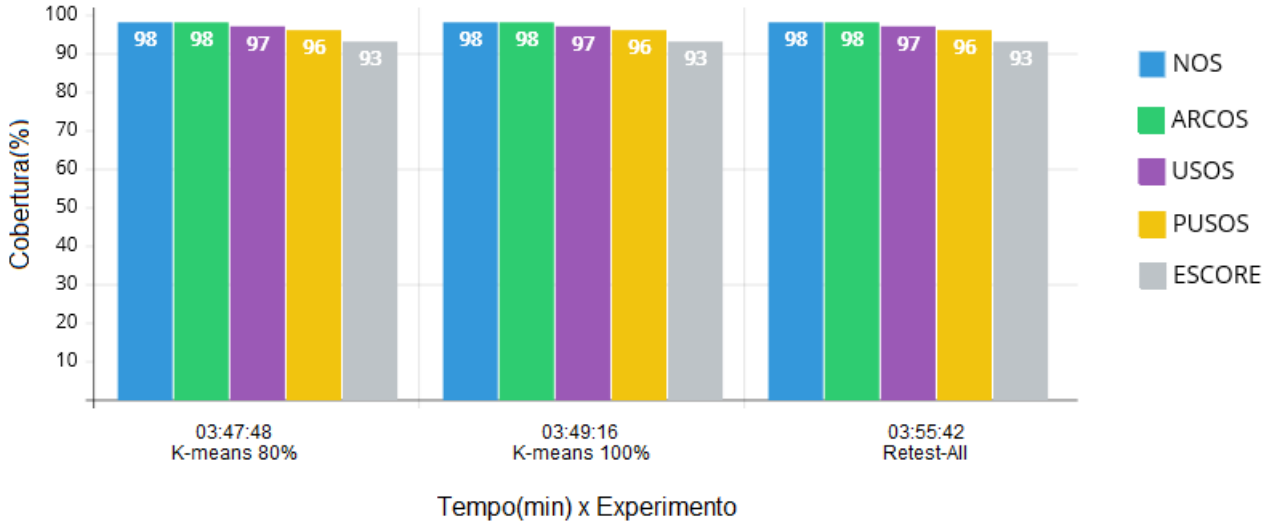
Figura 6.8 – Resultados do Experimentos do *MDC* e EM



Fonte – Elaborada pela autora.

No experimento de redução com o algoritmo K-means e o método *MDC*, o experimento com o fator de redução 50 não foi considerado na reexecução, dado que este não alcançou a quantidade casos de teste do conjunto mínimo. Porém, foi atingido os percentuais de cobertura dos critérios estruturais da abordagem *retest all* e foi reduzido o tempo de execução com os fatores de redução 80 e 100, como ilustrado na Figura 6.9.

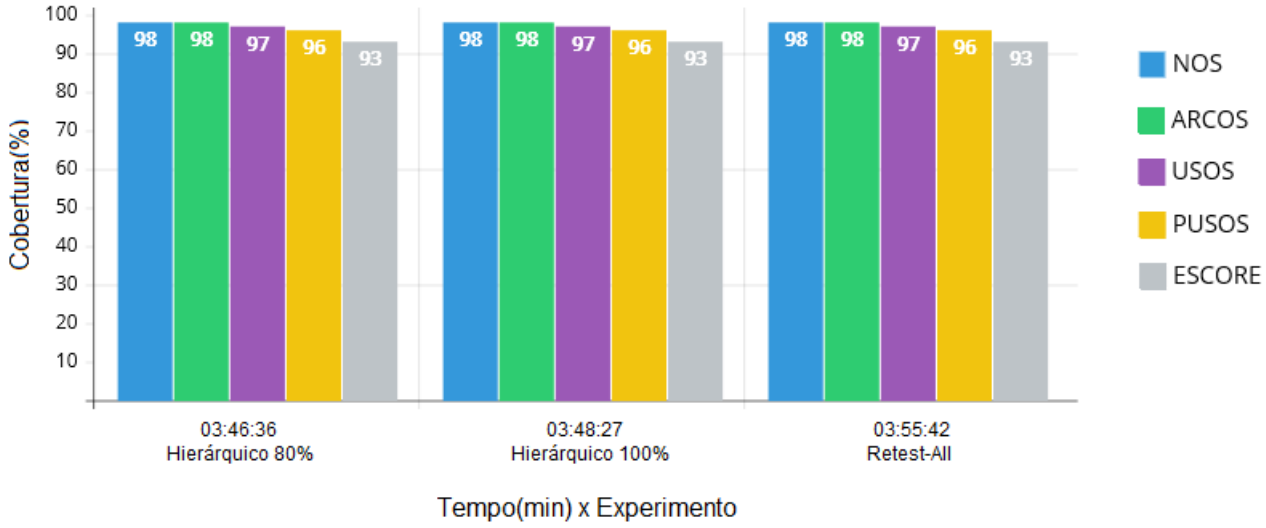
Figura 6.9 – Resultados do Experimentos do *MDC* e K-means



Fonte – Elaborada pela autora.

Assim como no experimento anterior, nos experimentos com o algoritmo Hierárquico e o método *MDC*, não foi considerado o experimento com o fator de redução 50, pois o mesmo também não atingiu a quantidade casos de teste do conjunto mínimo. Contudo, foi possível manter os percentuais de cobertura dos critérios estruturais da abordagem *retest all* com os fatores de redução 80 e 100. Esses resultados estão apresentados na Figura 6.10.

Figura 6.10 – Resultados do Experimentos do *MDC* e Hierárquico

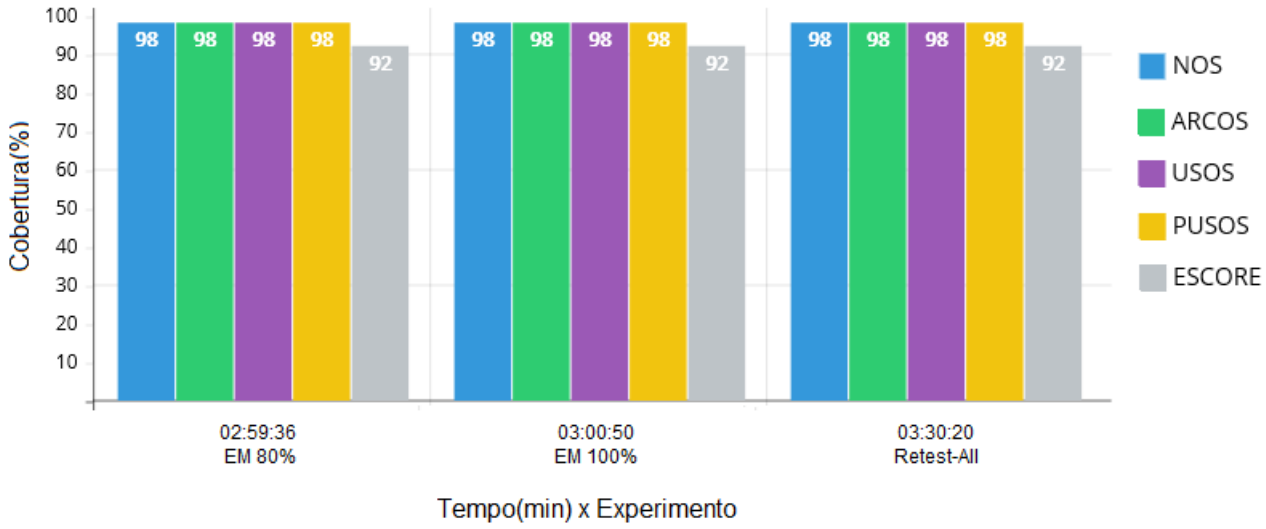


Fonte – Elaborada pela autora.

6.7.2 Análise de Resultados para o Método *Pirâmide*

A Figura 6.11 demonstra as porcentagens de cobertura dos critérios de teste e o tempo de execução alcançados pelos experimentos de redução com o algoritmo EM e o método *Pirâmide*. O experimento com o fator de redução de 50, não foi alcançou a quantidade casos de teste do conjunto mínimo, logo este não foi utilizado na reexecução. Para os fatores de redução 80 e 100, o algoritmo EM alcançou as porcentagens de cobertura dos critérios estruturais da abordagem retest all e reduziu o tempo de execução.

Figura 6.11 – Resultados do Experimentos do *Pirâmide* e EM

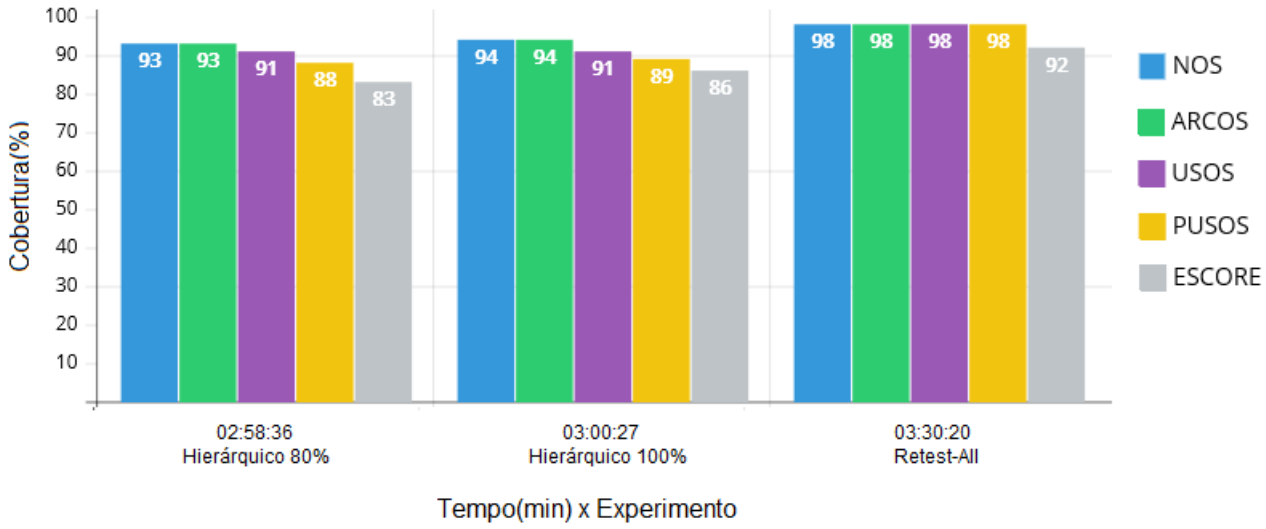


Fonte – Elaborada pela autora.

O experimento de redução com o algoritmo K-means e o método *Pirâmide* não foi possível ser realizado, pois nenhum dos experimentos realizado com os fatores de redução aplicados nesse trabalho alcançou a quantidade casos de teste do conjunto mínimo.

Para os experimentos com o algoritmo Hierárquico e o método *Pirâmide*, não foi possível manter os percentuais de cobertura dos critérios estruturais da abordagem *retest all* com os fatores de redução 80 e 100. Ressaltando, que o experimento com o fator de redução de 50 foi descartado, uma vez que este não alcançou a quantidade casos de teste do conjunto mínimo. Esses resultados estão apresentados na Figura 6.12.

Figura 6.12 – Resultados do Experimentos do *Pirâmide* e Hierárquico



Fonte – Elaborada pela autora.

6.8 CONSIDERAÇÕES FINAIS

Analisando os resultados apresentados nos gráficos dos experimentos com o método *MDC* e os algoritmos EM, K-Means e Hierárquico, conclui-se que é possível alcançar a redução desejada para diminuir o esforço do teste de regressão utilizando os FR 80 e 100. Desse modo, foi atingido o máximo possível de cobertura dos critérios de teste e, ao mesmo tempo, reduzido o tempo de execução. Além disso, infere-se que o fator de redução mais eficiente é o 80, dado que este mantém os percentuais de cobertura e tem a quantidade de casos de teste mais próxima do conjunto mínimo.

Já para os resultados expressados nos gráficos dos experimentos com o método *Pirâmide* e os algoritmos EM, K-Means e Hierárquico, percebe-se que a redução desejada para diminuir o esforço do teste de regressão foi alcançada apenas utilizando o EM com os FR 80 100. Apesar dos experimentos com o Hierárquico aplicando os FR 80 e 100 terem atingido a quantidade de casos de teste do conjunto mínimo, este não manteve os percentuais de cobertura da abordagem do *retest all*.

No caso dos experimentos com o K-means, não foi possível atingir nem a quantidade de casos de teste do conjunto mínimo. Isso ocorreu devido a maneira em que esses algoritmos distribuíram os casos de teste entre os *clusters*. Existiu uma discrepância considerável na quantidade de casos de testes contidos em cada *clusters*. Ao contrário do algoritmo EM, onde

esta distribuição foi realizada de forma balanceada. Devido a isso, é necessário um estudo mais aprofundado dos parâmetros utilizados para realizar os experimentos de redução com métodos mais complexo.

7 CONSIDERAÇÕES FINAIS

Neste capítulo, são relatadas as conclusões obtidas após a realização deste trabalho, assim como sugestões para trabalhos futuros.

7.1 CONCLUSÕES

Esse trabalhou descreveu o processo de análise, automatização e validação da metodologia apresentada por (SANTOS, 2016) que consistia em um método capaz de remover redundâncias em um conjunto de casos de teste sem perda significativa da cobertura dos critérios estruturais e escore de mutação em relação ao conjunto de teste original em programas da linguagem Java. Esta proposta de automatização deu-se devido ao fato de que a execução manual da metodologia estava passível de erros humanos. Além disso, em relação ao tempo, a execução de forma manual era uma atividade muito custosa e dependo da quantidade de casos de testes contido no conjunto original era até mesmo inviável.

Em virtude disso, o processo de realização da metodologia foi automatizado, desde da captura dos dados de execução da ferramentas Jabuti e Mujava até a identificação da quantidade de casos de teste que devem ser reexecutados para cada classe de equivalência durante a execução da atividade do teste de regressão. Assim, reduzindo o tempo gasto para a realização dessa atividade. Para isso, foi necessário integrar as ferramentas Jabuti, Mujava e Weka em um única ferramenta.

Após o processo de integração, foi implementado as seguintes funcionalidades: a geração do arquivo Arff com os dados de execução das ferramentas de teste, o acionamento dos algoritmos de agrupamento e classificação, assim como a interpretação das regras de classificação geradas, e também foi incluída a funcionalidade aplicar o cálculo da Progressão Aritmética (PA) para auxiliar o testador na redução do conjunto de casos de teste original.

Ao analisar os resultados obtidos por meio da aplicação da metodologia, pode-se concluir que o trabalho alcançou o objetivo proposto, considerando que a ferramenta elaborada foi capaz de identificar as classes de equivalência e definir a ordem de priorização das mesmas de forma correta. Além de que ao realizar novamente os testes estruturais e baseados em erro, percebeu-se que os novos conjuntos de teste definidos pelo método de redução não apresentaram perdas na porcentagem de cobertura dos critérios estruturais e do escore de mutação quando

foram utilizados os fatores de redução 80 e 100 para o método *MDC* e o método *Pirâmide*, e também ocorreu uma redução do tempo total da execução dos testes.

Nesse contexto, se levarmos em conta de que para métodos mais complexos do que os utilizados como objeto de validação nesta pesquisa, existirão uma quantidade ainda maior de casos de teste e, consequentemente, um maior tempo de execução. Logo, este método de redução pode apresentar resultados ainda mais significativos nesses casos. Por fim, é fundamental enfatizar que o fator de redução e os algoritmos de aprendizado de máquina aplicados na execução da metodologia estão diretamente associados à complexidade do programa. Por exemplo, para o método *MDC*, foi possível alcançar a porcentagem de cobertura dos critérios estruturais e do escore de mutação da abordagem *retest all* com os todos os algoritmos utilizados. Porém, para o método *Pirâmide* apenas o algoritmo EM alcançou o objetivo desejado. O código-fonte da ferramenta desenvolvida pode ser encontrado no repositório do Projeto AI+RTesting no Github (https://github.com/airtesting/airtesting).

7.1.1 Trabalhos Futuros

Ao longo desta pesquisa surgiram novas possibilidades que, devido a limitação de tempo, não foram implementadas. A primeira delas é uma análise mais precisa dos valores de aproximação para o valor da razão e dos elementos da PA. Atualmente, o cálculo está considerando três algarismos após a vírgula. Acredita-se que tal alteração poderia fazer com que a quantidade de casos de teste selecionada pelo método de redução fosse ainda mais próxima da quantidade de casos contidos no conjunto mínimo.

A outra possibilidade é a inclusão da reexecução dos novos conjuntos de casos de teste obtidos pela redução no processo automatizado. Esta atividade de reexecução ainda está sujeita ao erro humano, além de que é um processo trabalhoso. Por fim, também existe a possibilidade de um estudo mais aprimorado do modo em que é selecionado os casos de teste no conjunto redundante para serem reexecutados, tornando essa atividade mais otimizada e precisa. Assim, reduzindo ainda mais que o testador gasta para realizar o teste de regressão.

REFERÊNCIAS

ASCARI, L. C. Teste baseado em defeitos de classes java utilizando aspectos e mutação de especificações ocl. 2009.

CARVALHO, D. R. Árvore de decisão/algoritmo genético para tratar o problema de pequenos disjuntos em classificação de dados. **Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil. Doctor Thesis. 162pp**, 2005.

CRISTO, T. A. B. d. Ai+rtesting - um mÉtodo de seleÇÃo e de priorizaÇÃo para apoiar o teste de regressÃo utilizando aprendizado de mÁquina. 2017.

DASH, R.; DASH, R. Application of k-mean algorithm in software maintenance. v. 2, p. 442–445, 01 2012.

DAVIS, A. M. **201 Principles of Software Development**. New York, NY, USA: McGraw-Hill, Inc., 1995. ISBN 0-07-015840-1.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. [S.l.]: Elsevier, 2007.

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

DIJKSTRA, E. W. The humble programmer. **Commun. ACM**, v. 15, n. 10, p. 859–866, 10 1972.

FREITAS, A. A.; LAVINGTON, S. H. **Mining very large databases with parallel processing**. [S.l.]: Springer Science & Business Media, 1997. v. 9.

GRAVES, T. L.; HARROLD, M. J.; KIM, J.-M.; PORTER, A.; ROTHERMEL, G. An empirical study of regression test selection techniques. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 10, n. 2, p. 184–208, 2001.

HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WITTEN, I. H. The weka data mining software: an update. **ACM SIGKDD explorations newsletter**, ACM, v. 11, n. 1, p. 10–18, 2009.

HAN, J.; PEI, J.; KAMBER, M. **Data mining: concepts and techniques**. [S.l.]: Elsevier, 2011.

ISO/IEC. **ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models**. [S.l.], 2010.

JONES, J. A.; HARROLD, M. J. Test-suite reduction and prioritization for modified condition/-decision coverage. **IEEE Transactions on software Engineering**, IEEE, v. 29, n. 3, p. 195–209, 2003.

KUKOLJ, S.; MARINKOVIC, V.; POPOVIC, M.; BOGNÁR, S. Selection and prioritization of test cases by combining white-box and black-box testing methods. In: IEEE. **Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the**. [S.l.], 2013. p. 153–156.

LACHMANN, R.; SCHULZE, S.; NIEKE, M.; SEIDL, C.; SCHAEFER, I. System-level test case prioritization using machine learning. In: IEEE. **Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on**. [S.l.], 2016. p. 361–368.

LENZ, A. R. Utilizando técnicas de aprendizado de máquina para apoiar o teste de regressão. 2010.

LENZ, A. R. Utilizando técnicas de aprendizado de máquina para apoiar o teste de regressão. 2010.

MA, Y.-S.; OFFUTT, J. Description of method-level mutation operators for java. **Electronics and Telecommunications Research Institute, Korea, Tech. Rep.**, 2005.

MAFRA, J.; MIRANDA, B.; IYODA, J.; SAMPAIO, A. Test case selector: Uma ferramenta para seleç ao de testes. 2009.

MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S.; JINO, M. Introduçao ao teste de software. **Rio de Janeiro: Campus**, 2007.

MCLACHLAN, G.; KRISHNAN, T. **The EM algorithm and extensions**. 2. ed. ed. Hoboken, NJ: Wiley, 2008. (Wiley series in probability and statistics). ISBN 978-0-471-20170-0.

MITCHELL, T. M. **Machine Learning**. 1. ed. New York, NY, USA: McGraw-Hill, Inc., 2014. ISBN 0070428077, 9780070428072.

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. **Sistemas Inteligentes-Fundamentos e Aplicações**, v. 1, n. 1, 2003.

MYERS, G. J. **Art of Software Testing**. 3. ed. [S.l.: s.n.], 2011.

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

NAGAR, R.; KUMAR, A.; SINGH, G. P.; KUMAR, S. Test case selection and prioritization using cuckoos search algorithm. In: IEEE. **Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), 2015 International Conference on**. [S.l.], 2015. p. 283–288.

NETO, A. C. D. Introdução a teste de software. **Engenharia de Software Magazine**, v. 1, p. 22, 2007.

OLIVEIRA, A. A. L. d. et al. Uma abordagem coevolucionária para seleção de casos de teste e programas mutantes no contexto do teste de mutação. Universidade Federal de Goiás, 2013.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. [S.l.]: Palgrave Macmillan, 2005.

PRESSMAN, R. S. **Software engineering: a practitioner's approach**. 8. ed. [S.l.]: McGraw-Hill, 2016.

ROCHA, A. R. C. d.; MALDONADO, J. C.; WEBER, K. C. Qualidade de software. **Ana Regina, José Carlos Maldonado, Kival Weber-São Paulo: Pretice Hall**, 2001.

ROTHERMEL, G.; ELBAUM, S.; MALISHEVSKY, A. G.; KALLAKURI, P.; QIU, X. On test suite composition and cost-effective regression testing. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 13, n. 3, p. 277–331, 2004.

SAIFAN, A. A.; ALSUKHNI, E.; ALAWNEH, H.; SBAIH, A. A. Test case reduction using data mining classifier techniques. **Int. J. Softw. Innov.**, v. 4, n. 4, p. 56–70, 03 2016. ISSN 2166-7160.

SANTOS, E. P. dos. Ai+rtesting: Remoção de redundâncias de conjuntos de casos de teste a partir de dados de execução do teste de mutação. 2018.

SANTOS, R. B. E. **Redução de Conjuntos de Casos de Teste aplicados ao Teste de Regressão em programas JAva utilizando algoritmos de Aprendizado de Máquina**. Universidade do Estado da Bahia, 2016. Disponível em: <http://www.csi.uneb.br/csitcc/files/ArqTrabCien-22930408.pdf>.

SIMONS, C. **PRIORIZAÇÃO DE CASOS DE TESTES DE REGRESSÃO USANDO AMOSTRAGEM POR PERSEGUIÇÃO DE DEFEITOS**. Tese (Doutorado) — Pontifícia Universidade Católica do Paraná, 2010.

SOMMERVILLE, I.; MELNIKOFF, S. S. S.; ARAKAKI, R.; BARBOSA, E. de A. **Engenharia de software**. [S.l.]: Addison Wesley São Paulo, 2011. v. 9.

VINCENZI, A. M. R. **Orientação a objeto: definição, implementação e análise de recursos de teste e validação**. Tese (Doutorado) — Universidade de São Paulo (USP). Instituto de Ciências Matemáticas e de Computação de São Carlos, 2004.

WITTEN, I. H.; FRANK, E.; HALL, M. A.; PAL, C. J. **Data Mining: Practical machine learning tools and techniques**. [S.l.]: Morgan Kaufmann, 2016.

YOO, S.; HARMAN, M.; TONELLA, P.; SUSI, A. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: ACM. **Proceedings of the eighteenth international symposium on Software testing and analysis**. [S.l.], 2009. p. 201–212.