UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

GUSTAVO DA MOTA RAMOS

# Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos

São Paulo
2018

GUSTAVO DA MOTA RAMOS

# Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos

Versão original

Dissertação apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação.

Área de concentração: Metodologia e Técnicas da Computação

Orientador: Prof. Dr. Marcelo Medeiros Eler

São Paulo

2018

CATALOGAÇÃO-NA-PUBLICAÇÃO
(Universidade de São Paulo. Escola de Artes, Ciências e Humanidades. Biblioteca)
CRB-8 4936

Ramos, Gustavo da Mota
Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos / Gustavo da Mota Ramos ; orientador, Marcelo Medeiros Eler. – 2018.
75 p. : il.

Dissertação (Mestrado em Ciências) - Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo.
Versão original

1. Teste e avaliação de software. 2. Algoritmos genéticos. 3. Métricas de software. 4. Método de desenvolvimento de software orientado a objeto. I. Eler, Marcelo Medeiros, orient. II. Tìtulo.

CDD 22.ed.– 005.14

Dissertação de autoria de Gustavo da Mota Ramos, sob o título **"Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos"**, apresentada à Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, para obtenção do título de Mestre em Ciências pelo Programa de Pós-graduação em Sistemas de Informação, na área de concentração Metodologia e Técnicas da Computação, aprovada em _____ de _____ de _____ pela comissão julgadora constituída pelos doutores:

_____

**Prof. Dr.** _____

Instituição: _____

Presidente

_____

**Prof. Dr.** _____

Instituição: _____

_____

**Prof. Dr.** _____

Instituição: _____

_____

**Prof. Dr.** _____

Instituição: _____

*Aos meus pais, Nivaldo e Izildinha, que não mediram esforços para que eu chegasse até aqui. Eles são responsáveis pela maior herança da minha vida: meus estudos.*

## Resumo

RAMOS, Gustavo da Mota. **Seleção entre estratégias de geração automática de dados de teste por meio de métricas estáticas de softwares orientados a objetos**. 2018. 75p f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2018.

Produtos de software com diferentes complexidades são criados diariamente através da elicitação de demandas complexas e variadas juntamente a prazos restritos. Enquanto estes surgem, altos níveis de qualidade são esperados para tais, ou seja, enquanto os produtos tornam-se mais complexos, o nível de qualidade pode não ser aceitável enquanto o tempo hábil para testes não acompanha a complexidade. Desta maneira, o teste de software e a geração automática de dados de testes surgem com o intuito de entregar produtos contendo altos níveis de qualidade mediante baixos custos e rápidas atividades de teste. Porém, neste contexto, os profissionais de desenvolvimento dependem das estratégias de geração automáticas de testes e principalmente da seleção da técnica mais adequada para conseguir maior cobertura de código possível, este é um fator importante dados que cada técnica de geração de dados de teste possui particularidades e problemas que fazem seu uso melhor em determinados tipos de software. A partir desde cenário, o presente trabalho propõe a seleção da técnica adequada para cada classe de um software com base em suas características, expressas por meio de métricas de softwares orientados a objetos a partir do algoritmo de classificação *Naive Bayes*. Foi realizada uma revisão bibliográfica de dois algoritmos de geração, algoritmo de busca aleatório e algoritmo de busca genético, compreendendo assim suas vantagens e desvantagens tanto de implementação como de execução. As métricas CK também foram estudadas com o intuito de compreender como estas podem descrever melhor as características de uma classe. O conhecimento adquirido possibilitou coletar os dados de geração de testes de cada classe como cobertura de código e tempo de geração a partir de cada técnica e também as métricas CK, permitindo assim a análise destes dados em conjunto e por fim execução do algoritmo de classificação. Os resultados desta análise demonstraram que um conjunto reduzido e selecionado das métricas CK é mais eficiente e descreve melhor as características de uma classe se comparado ao uso do conjunto por completo. Os resultados apontam também que as métricas CK não influenciam o tempo de geração dos dados de teste, entretanto, as métricas CK demonstraram correlação moderada e influência na seleção do algoritmo genético, participando assim na sua seleção pelo algoritmo *Naive Bayes*.



Palavras-chaves: Métricas CK. Teste de software. Geração de testes. Cobertura de testes. Naive bayes. Algoritmo genético.

**Abstract**

RAMOS, Gustavo da Mota. **Selection between whole test generation strategies by analysing object oriented software static metrics** . 2018. 75 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, DefenseYear.

Software products with different complexity are created daily through analysis of complex and varied demands together with tight deadlines. While these arise, high levels of quality are expected for such, as products become more complex, the quality level may not be acceptable while the timing for testing does not keep up with complexity. In this way, software testing and automatic generation of test data arise in order to deliver products containing high levels of quality through low cost and rapid test activities. However, in this context, software developers depend on the strategies of automatic generation of tests and especially on the selection of the most adequate technique to obtain greater code coverage possible, this is an important factor given that each technique of data generation of test have peculiarities and problems that make its use better in certain types of software. From this scenario, the present work proposes the selection of the appropriate technique for each class of software based on its characteristics, expressed through object oriented software metrics from the naive bayes classification algorithm. Initially, a literature review of the two generation algorithms was carried out, random search algorithm and genetic search algorithm, thus understanding its advantages and disadvantages in both implementation and execution. The CK metrics have also been studied in order to understand how they can better describe the characteristics of a class. The acquired knowledge allowed to collect the generation data of tests of each class as code coverage and generation time from each technique and also the CK metrics, thus allowing the analysis of these data together and finally execution of the classification algorithm. The results of this analysis demonstrated that a reduced and selected set of metrics is more efficient and better describes the characteristics of a class besides demonstrating that the CK metrics have little or no influence on the generation time of the test data and on the random search algorithm . However, the CK metrics showed a medium correlation and influence in the selection of the genetic algorithm, thus participating in its selection by the algorithm naive bayes.

Keywords: CK metrics. Software testing. Test data generation. Code coverages. Naive bayes. Genetic algorithm.

## Lista de figuras

## Lista de algoritmos

# Lista de quadros

## Lista de tabelas

## Lista de abreviaturas e siglas

| | |
|---|---|
| CBO | *Coupling between objects* |
| CK | *Chidamber & Kemerer* |
| CUT | *Class under test* |
| DIT | *Depth inheritance tree* |
| LCOM | *Lack of cohesion of methods* |
| LOC | *Lines of code* |
| NOC | *Number of children* |
| NOF | *Number of fields* |
| NOM | *Number of methods* |
| NOPF | *Number of public fields* |
| NOPM | *Number of public methods* |
| RFC | *Response for a Class* |
| SUT | *Software under test* |
| WMC | *Weight method class* |

**Sumário**

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

# 1 Introdução

Produtos de software de diferentes tamanhos e complexidades são utilizados todos os dias nos mais diversos contextos. Em qualquer circunstância, entretanto, a falta de qualidade nestes produtos pode caracterizar uma situação preocupante para seus produtores (HILBURN; TOWHIDNEJAD, 2002) (BINDER, 1994), uma vez que cada vez mais os níveis aceitáveis de qualidade de um software estão aumentando, tanto no que se refere ao seu comportamento observável externamente quanto ao seu processo de desenvolvimento e estrutura interna (NORTHROP, 2006) (BASHIR; BANURI, 2008) (GRAHAM; VEENENDAAL; EVANS, 2008).

O teste de software (TAHIR; MACDONELL; BUCHAN, 2014) é a maneira mais popular de verificar se um software atende às especificações descritas e cumpre o papel desejado pelos interessados (SOMMERVILLE *et al.*, 2008). Ele consiste na execução do programa sob teste para revelar seus defeitos. Para isso, casos de teste são gerados de maneira que satisfaçam a diversos critérios, geralmente baseados na especificação e na implementação do software (PEZZÈ; YOUNG, 2008).

Entretanto, gerar casos de teste para testar o software nos mais diversos contextos e dados de entrada é uma das atividades mais custosas no ciclo de vida de software, requisitando muito tempo e esforço para o seu planejamento, execução e manutenção. (TAHIR; MACDONELL; BUCHAN, 2014). Consequentemente, pesquisadores e profissionais da área passaram então a desenvolver abordagens para gerar casos de teste automaticamente e assim reduzir o custo e a complexidade desta tarefa. Diversas técnicas já foram utilizadas com teste objetivo: geração de testes aleatórios (PACHECO; ERNST, 2007); execução simbólica (CADAR; SEN, 2013); teste baseado em modelos (DICK; FAIVRE, 1993); e teste baseado em buscas (MCMINN, 2004; HARMAN; MANSOURI; ZHANG, 2012). Diversas ferramentas já foram desenvolvidas utilizando uma ou mais dessas técnicas, como, por exemplo, Randoop (PACHECO; ERNST, 2007), DART (GODEFROID; KLARLUND; SEN, 2005), CUTE (SEN; MARINOV; AGHA, 2005), JCute (SEN; AGHA, 2006), Klee (CADAR; DUNBAR; ENGLER, 2008), JavaPathFinder (VISSER; PăSăREANU; KHURSHID, 2004), PEX (TILLMANN; HALLEUX, 2008) e EvoSuite (FRASER; ARCURI, 2011).

Em particular, a técnica de teste baseado em busca tem sido amplamente utilizada para a geração de casos de teste. Neste contexto, destaca-se a ferramenta EvoSuite que

é capaz de gerar conjuntos de casos de teste para programas escritos em Java automaticamente. Ela utiliza um algoritmo genético para selecionar os melhores dados de teste e produzir os conjuntos de casos de teste que atingem altos níveis de cobertura estrutural e escore de mutação (FRASER; ARCURI, 2011). Esta ferramenta já foi extensivamente avaliada no que se refere à eficácia e escalabilidade (FRASER; ARCURI, 2013; ROJAS *et al.*, 2017; FRASER *et al.*, 2015; FRASER; ARCURI, 2014), e recentemente foi a ferramenta que obteve a maior pontuação na competição de ferramentas de geração de teste unitários promovida anualmente pelo Workshop on Search-Based Software Testing (FRASER *et al.*, 2017).

Recentemente, novos algoritmos evolutivos foram adicionados à ferramenta EvoSuite como alternativa ao algoritmo genético padrão para a geração de casos de teste (CAMPOS *et al.*, 2017): algoritmo genético monotônico, algoritmo genético de regime permanente (do inglês, steady state), 1 + ($\lambda$,$\lambda$), $\mu + \lambda$, MOSA (Many-Objective Sorting Algorithm) e DynaMOSA (dynamic MOSA); além da geração aleatoriamente já implementada anteriormente. Um estudo comparativo utilizando 346 classes selecionadas randomicamente de um conjunto de 117 projetos de código aberto mostrou que a escolha do algoritmo utilizado na geração de casos de teste pode influenciar os resultados (CAMPOS *et al.*, 2017).

## *1.1 Problema de pesquisa*

O estudo realizado para comparar a eficácia dos algoritmos na geração de casos de teste implementados na ferramenta EvoSuite mostrou que a escolha do algoritmo importa. Entretanto, nenhuma análise mais criteriosa foi realizada para identificar em que situações ou para que tipo de classe um algoritmo é melhor do que outro. Portanto, as seguintes questões de pesquisa emergem neste contexto:

- Existe alguma diferença de desempenho entre os algoritmos de geração de dados de teste: aleatório e algoritmos genéticos ?
- Alguma característica ou conjunto de características do software sob teste que faz com que uma técnica usada na geração de casos de teste seja melhor do que outra?
- É possível escolher entre dois algoritmos de geração de dados de testes a partir de classificador e métricas de softwares orientados e objetos ?

## *1.2 Objetivos*

Neste contexto, o objetivo geral deste projeto de pesquisa é definir um mecanismo para escolher o melhor algoritmo de geração de casos de testes de acordo com o problema em mãos.

Desta forma, ao invés de utilizar um único algoritmo para gerar todos os casos de teste de um projeto, os casos de teste de cada classe podem ser gerados por diferentes algoritmos dependendo das suas características.

Os objetivos específicos deste projeto de pesquisa são os seguintes:

1. Selecionar métricas de softwares orientados a objetos que possuem maior probabilidade de influenciar os resultados dos algoritmos na geração de casos de teste. Essa seleção será realizada utilizando-se a correlação entre as métricas e os resultados obtidos pelos algoritmos evolutivos em termos de cobertura de código dos casos de teste gerados por eles.
2. Utilizar um algoritmo de reconhecimento de padrões para analisar as métricas de uma classe de um software orientado a objetos e classificá-la de acordo com o algoritmo de geração de testes mais adequado.

## *1.3 Justificativa*

Os diversos algoritmos e abordagens de geração de casos de teste encontrados no ambiente acadêmico e industrial possuem pontos fracos e fortes. Em especial, a técnica de teste baseado em busca tem sido amplamente utilizada para gerar casos de teste, e muitos dos seus algoritmos conseguem convergir para resultados ótimos mais rapidamente do que os outros, mas não conseguem tratar de situações mais complexas, por exemplo. Portanto, utilizar o algoritmo que consegue obter os melhores resultados e convergir mais rapidamente para cada classe do projeto a ser testado irá aumentar consideravelmente a eficiência e a eficácia de ferramentas de geração de casos de teste como a EvoSuite, por exemplo. Além disso, os resultados obtidos podem encorajar a implementação de algoritmos evolutivos que são utilizados em outras áreas mas que ainda não utilizados na geração de casos teste.

## *1.4 Método de pesquisa*

Este projeto será, conduzido usando um método indutivo com objetivos de caráter exploratório, descritivo e experimental. Mais detalhes dos materiais e métodos de pesquisa utilizados neste projeto de pesquisa estão apresentados no Capítulo 6.

## *1.5 Organização do documento*

Esta proposta de pequisa está organizada da seguinte forma: No Capítulo 2 são apresentados os conceitos mais importantes e econômicos de motivação para o contexto de teste de software. Nos capítulos 3, 4 e 5 serão apresentados conceitos mais importantes que dão base para este projeto através da revisão bibliográfica construída. No capítulo 6 serão detalhados os materiais e métodos utilizados, os quais serão detalhados como uso no capítulo 7 e concluídos no capítulo 8.

# 2 Aspectos conceituais e econômicos

No presente capítulo será apresentado uma síntese dos conceitos base com o objetivo de fundamentação teórica dos capítulos seguintes e auxílio na estrutura do presente documento.

## *2.1 Teste de software*

O teste de software é uma subárea da engenharia de software que compreende as atividades capazes de determinar se um software em análise contém erros (GOODENOUGH; GERHART, 1975) com objetivo encontrar falhas e não o objetivo de provar a corretitude de software. Os testes não são capazes de demonstrar se um software em análise está livre de defeitos ou que irá se comportar como esperado em todas as circunstâncias, ou seja, testes são responsáveis somente por evidenciar a presença de possíveis erros e não a falta destes (PFLEEGER; ATLEE, 2010).

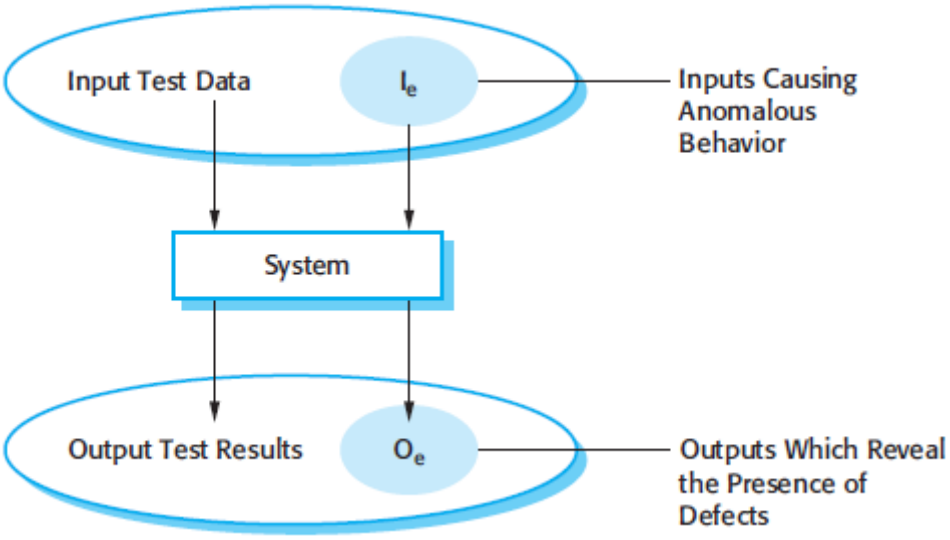
A partir de testes é possível validar se um software cumpre o propósito de desenvolvimento e por fim eliminar possíveis defeitos e assim garantir maior qualidade de *software* (SOMMERVILLE, 2010). De acordo com (PRESSMAN, 2009) e (DAVIS, 1995) são notórios os princípios de teste:

1. Todos os testes devem ser rastreáveis aos requisitos de usuários;
2. Testes devem ser planejados antes de sua execução;
3. O princípio de pareto deve ser aplicado ao teste de software;
4. Testes devem começar as menores unidades a fim de atingir partes maiores de software;
5. Testes exaustivos não são possíveis.

O fluxo de testes é composto por modelar o requisito em formato de casos de teste, contendo os detalhes de execução, dados de entradas e estado esperado do *SUT* (*Software under test*). Após a criação do caso de teste, este pode ser utilizado para execução, para executar um caso de teste, é necessário simular a execução do *SUT* e validar, a partir dos resultados extraídos da execução, possíveis anomalias e erros, testes de software são executados em grande parte dos casos com dados artificais (SOMMERVILLE, 2010). Com

os dados de saída é possível verificar se o estado encontrado é o mesmo esperado, este modelo é demonstrado na figura 1.

Figura 1 – Modelo de teste de software




Fonte: (SOMMERVILLE, 2010)

### 2.1.1 Caso de teste

Um caso de teste é um conjunto de possíveis entradas, condições de execução e resultados esperados de acordo com seus objetivos, estes quando agrupados compõe uma instância única de execução em termos de entrada, execução e resultados esperados, estas saídas quando analisadas em relação aos dados esperados após execução determinam se o estado encontrado pela aplicação é o esperado (SINGH, 2014). Os casos de teste são projetados para cenários particulares de teste aos quais é desejado comparar estados e resultados coletados a partir do resultado de sua execução com requisitos funcionais anteriormente definidos (JACOB; PRASANNA, 2016).

### 2.1.2 *Mocks*

Ao construir testes automatizados é comum testar componentes de softwares que dependam de outros softwares ou inclusive de outros componentes do mesmo software, e desenvolvedores normalmente necessitam testar o software com todas suas dependências ou simulações (SPADINI *et al.*, 2017). Ao testar todas as dependências juntamente,

desenvolvedores passam a ter um comportamento similar ao ambiente real de produção (SPADINI *et al.*, 2017).

Apesar dos benefícios, algumas dependências como bancos de dados e chamadas a serviços *online* podem ser custosos para serem testados, nestes casos desenvolvedores podem usar *mocks* (SPADINI *et al.*, 2017). *Mocks* são usados para substituir dependências reais em softwares ao simular suas características mais importantes, usualmente métodos em *mocks* retornam valores desejados de acordo com os valores de entrada (SPADINI *et al.*, 2017).

### 2.1.3 Testes manuais

O teste manual é aquele executado por um profissional de qualidade, o qual geralmente possui um caso de teste no qual são descritos os passos de execução e por fim, o responsável toma decisões de acordo com a finalização da execução (KAPROCKI; PEKOVIC; VELIKIC, 2015). Espera-se que os testes projetados para aplicações sejam, em geral, compostos pelos seguintes itens:

- Estado inicial do teste;
- Entradas para uso;
- Sequência de passos para execução;
- Uma ou mais saídas esperadas.

### 2.1.4 Testes automatizados

Os testes automatizados possuem uma estrutura comum aos testes manuais, porém, o maior diferencial deste tipo de teste é delegar para a máquina a execução e possivelmente a validação dos resultados obtidos. Apesar de testes automatizados possuírem diversos papéis, como testes funcionais, testes de stress, é notório que testes automatizados têm a capacidade de executar uma grande quantidade de casos de teste em um curto período de tempo, com maior precisão (KAPROCKI; PEKOVIC; VELIKIC, 2015).

## *2.2 Teste caixa preta*

O teste caixa preta é uma categoria de testes funcionais o qual na sua execução não existe o conhecimento da estrutura interna do SUT (XU *et al.*, 2016), o conceito nasceu da ideia que a estrutura interna do SUT está coberta por uma caixa preta de maneira não se tenha conhecimento de sua estrutura interna. Este conceito foca somente no funcionamento do *SUT*, desta maneira o profissional concentra no que o software deve fazer e elimina o raciocínio baseado na maneira como é feito (JACOB; PRASANNA, 2016).

Geralmente ao efetuar um teste caixa preta, o profissional de qualidade ou ferramentas de execução automatizada de scripts de teste poderão interagir com a interface de usuário de maneira que possam fornecer dados de entrada sem quaisquer noções da lógica computacional por trás da interface utilizada (XU *et al.*, 2016). Idealmente este tipo de teste é guiado por especificações formais dos requisitos elicitados ou modelos de dados de teste, porém na prática estas especificações nem sempre estão disponíveis (WALKINSHAW; FRASER, 2017).

## *2.3 Qualidade de software e aspectos econômicos*

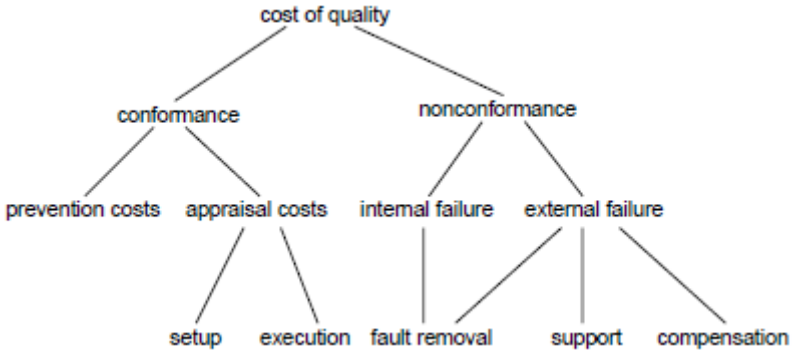
A Garantia de Qualidade ou *Quality Assurance* é considerada uma das partes mais caras do desenvolvimento de software (WAGNER; SEIFERT, 2005), o conceito de garantia de qualidade tornou-se mais evidente após a primeira conferência de engenharia de software com o propósito de estabelecer melhores princípios e estratégias econômicas para atingir um estado viável de software. Nesta conferência de 1968 discutiu-se sobre a disciplina de teste e controle de qualidade em diversas fases de desenvolvimento de software desde então (REPASI, 2009).

De fato custos de garantia de qualidade são elevados (WAGNER; SEIFERT, 2005) (KOREL, 1990), eles consistem em todo e qualquer valor investido em atividades cujo propósito almeja qualidade de software (PRESSMAN, 2009). Entretanto é notável que investimentos em melhoria de qualidade nem sempre são bem aceitos por todos os interessados no desenvolvimento de softwar; é comum a falta de compreensão sobre o retorno de investimento e o valor de negócio, ambos gerados a partir de atividades relacionadas a qualidade de desenvolvimento de software (SLAUGHTER; HARTER; KRISHNAN, 1998).

Apesar da falta de compreensão, os investimentos em qualidade de software são importantes e devem ser planejados, dado que cada valor gasto em horas de trabalho e não investido em retrabalho pode ser usado para melhorias rápidas e criação de novas funcionalidades (SLAUGHTER; HARTER; KRISHNAN, 1998).

Os custos direcionados à qualidade de software são categorizados em *conformance* e *nonconformance* (SLAUGHTER; HARTER; KRISHNAN, 1998)(PRESSMAN, 2009) como demonstrado na figura a seguir:

Figura 2 – Hierarquia da classificação dos tipos de custo presentes em desenvolvimento de software




Fonte: (WAGNER; SEIFERT, 2005)

Os custos de *conformance* caracterizam os valores com o intuito de atingir maior qualidade de produto, este por sua vez é divido em custo de prevenção e custo de avaliação. (WAGNER; SEIFERT, 2005)

Custos de prevenção são aqueles associados com o intuito de prevenir defeitos antes que possam ocorrer, geralmente são compreendidos por treinamentos, equipamentos, revisões técnicas formais, atividades de planejamento de qualidade e revisões de produto (WAGNER; SEIFERT, 2005) (PRESSMAN, 2009).

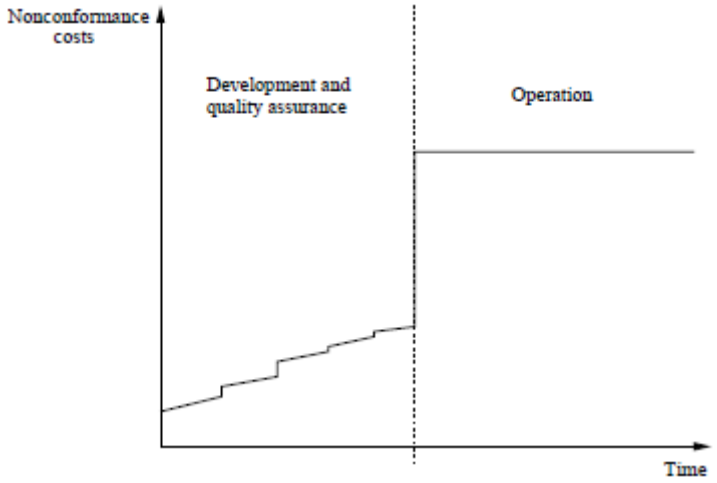
Custos de avaliação compreendem custos relacionados a medidas e extração de métricas, avaliação e auditoria de produtos.(WAGNER; SEIFERT, 2005), incluem atividades para ganhar conhecimento da condição do software em análise para cada início de ciclo, incluem: avaliações de processo e entre processos e manutenção (PRESSMAN, 2009).

Custos de não conformidade são os custos relacionados aos cenários em que não seguem o planejado e defeitos produzindo um erro e por fim levando à falha (WAGNER; SEIFERT, 2005), nesta categoria estão as subcategorias de custos de falha externa e custos de falha interna (PRESSMAN, 2009).

Os custos de falha externa caracterizam os custos associados aos defeitos após liberação do produto de software para uso, são exemplos: resolução de chamados, retorno e substituição de produtos e garantias (PRESSMAN, 2009) enquanto os custos de falha interna são os custos aplicados para remoção de falhas antes de liberação para uso.

O custo relativo para encontrar e reparar um defeito aumenta consideravelmente na linha do tempo entre as fases iniciais e finais do ciclo de desenvolvimento, sendo assim, custos do tipo *nonconformance* se tornam mais caros ao longo do tempo de projeto, como demonstrado a seguir (WAGNER; SEIFERT, 2005):

Figura 3 – Crescimento dos custos do tipo *nonconformance* ao longo do tempo de projeto




Fonte: (WAGNER; SEIFERT, 2005)

Apesar do custo relativo para encontrar e reparar um defeito aumenta consideravelmente na linha do tempo, o processo de desenvolvimento de um software flui entre os diversos ciclos, e por consequência, diferentes tipos de investimentos são feitos com o intuito de garantiar maiores níveis de qualidade (PRESSMAN, 2009).

# 3 Geração automática de dados de teste

A manutenção de softwares é usualmente vista não somente como uma tarefa custosa mas dispendiosa, as atividades ligadas a esta somam por volta de 60% do investimento total de um software (SHAMSHIRI *et al.*, 2018). Concomitantemente os profissionais de desenvolvimento de software necessitam testar o SUT utilizando-se de casos de teste que revelem falhas, consumindo assim mais tempo e recursos. (DAVE; AGRAWAL, 2015).

O alto custo presente em testes de mutação e geração de mutantes pode ser reduzido pela geração automática de dados de teste (DAVE; AGRAWAL, 2015), a geração automática de testes poupa o tempo investido por desenvolvedores na escrita de testes unitários (SHAMSHIRI *et al.*, 2018).

A geração automática de dados de teste consiste em uma tarefa importante no processo de testes de software, ela possui o objetivo de melhorar a qualidade de um *SUT* com o intuito de aumentar a qualidade do software e diminuir o custo investido (SHAMSHIRI *et al.*, ). A geração automática de dados de teste possui também um papel importante no processo de desenvolvimento de software pois pode afetar diretamente a qualidade final de um produto de software (DAVE; AGRAWAL, 2015).

Os testes gerados podem ser incluídos nos repositórios de código do SUT como qualquer outro artefato de software (SHAMSHIRI *et al.*, 2018), porém conjuntos de testes volumosos podem levar mais tempo para serem executadas e precisam ser avaliados antes de serem incluídos em uma suíte de testes (SHAMSHIRI *et al.*, ). Ferramentas como a EvoSuite têm se mostrado eficazes em atingir alta cobertura de código em diferentes tipos de softwares orientados a objetos. (CAMPOS *et al.*, 2017)

A maneira mais fácil de avaliar conjuntos de testes automaticamente gerados é avaliar sua capacidade de expor falhas (SHAMSHIRI *et al.*, 2018), as avaliações de conjuntos de testes automaticamente gerados ocorre por meio de métricas como cobertura de linhas de código, cobertura de caminhos ou estimativas da habilidade de detecção de falhas, sendo um exemplo deste o *score* de mutação (SHAMSHIRI *et al.*, 2018).

## *3.1 Algoritmos baseados em busca*

Algoritmos baseados em busca têm se mostrado efetivos na geração de dados de teste quando otimizados para cobertura de código de diferentes tipos de software (CAMPOS *et al.*, 2017). Os algoritmos baseados em busca também mostraram-se eficazes quando aplicada para geração de suítes de testes unitários otimizados para cobertura de código em classes de softwares orientados a objetos (CAMPOS *et al.*, 2017).

Novas técnicas de busca foram desenvolvidas nos últimos anos (ROJAS; FRASER, 2017), algoritmos de busca estão entre as técnicas mais bem sucedidas para automaticamente gerar dados de teste e testes unitários (ROJAS; FRASER, 2017). Técnicas como buscas aleatórias, buscas locais, *simulated annealing* e algoritmos genéticos são algumas das técnicas mais utilizadas. (ROJAS; FRASER, 2017)

Uma abordagem popular é o uso de algoritmos evolutivos no qual os indivíduos utilizados na população são caracterizados por testes os quais após gerados compõe suítes de testes completas, a construção destas suítes ocorre com objetivo de encontrar suítes de teste os quais atingem a cobertura de código máxima a partir de técnicas baseadas em evolução natural (CAMPOS *et al.*, 2017)

## *3.2 Algoritmo baseado em busca aleatória*

O algoritmo de busca aleatória, ou *random search*, é um algoritmo o qual um mecanismo de seleção deve ser estabelecido para escolher os candidados em um espaço de tamanho $X$, o processo aleatório pode ser definido como a função de densidade de probabilidade $P(x)$ a qual expressa a probabilidade de qualquer candidato presente em um espaço $X$ de possíveis soluções (KARNOPP, 1963). Apesar de estudos na área, existe um grande uso de técnicas padrão, técnicas como estas geralmente baseiam-se em um gerador aleatório entre 0 e 1, o qual a saída pode ser utilizada para produzir uma aproximação da distribuição desejada. (KARNOPP, 1963)

O algoritmo de busca aleatória é um algoritmo que não utiliza operações de *crossover*, mutações ou selecões, ao contrário dos algoritmos genéticos seu funcionamento baseia-se puramente em estratégias de substituição (CAMPOS *et al.*, 2017). O algoritmo de busca aleatória consiste em repetidamente gerar amostras de candidatos presentes em um espaço

de busca de soluções possíveis para o problema a qual o candidato já presente na solução parcial é substituído se a aptidão do indivíduo gerado supera a aptidão do indivíduo já presente na resolução do problema em análise. (CAMPOS *et al.*, 2017)

O Algoritmo *Random testing* é uma variação dos algoritmo baseado em busca aleatória que utiliza o algoritmo de busca aleatória como base para construir suítes de testes de maneira incremental (CAMPOS *et al.*, 2017). O Random testing diferencia-se de outros algoritmos por gerar sequencias aleatórias de casos de teste juntamente aos dados de entrada também aleatoriamente gerados (SHAMSHIRI *et al.*, ), se um novo caso de teste gerado cobre um novo ramo do $SUT$ , o qual não foi coberto nas iterações anteriores, o novo caso de teste é adicionado ao conjunto (SHAMSHIRI *et al.*, ).

Os casos de testes são tradados como amostras individuais, se uma amostra gerada aumenta a cobertura de código, a amostra recém gerada é então inserida na suíte de testes, caso contrário é descartada (CAMPOS *et al.*, 2017). Segundo (KARNOPP, 1963) técnicas de busca aleatórias possuem alguns vantagens de utilização como:

- Facilidade de desenvolvimento: Por sua facilidade de programação do algoritmo, bons resultados podem ser obtidos com pouco esforço de desenvolvimento para resoluções de problemas de busca.
- Baixo custo: Os valores aleatórios utilizados em buscas aleatórias podem ser gerados com pouco custo computacional, muitos dos métodos necessitam meios de armazenamento e operações de comparação mutualmente simples.
- Eficiência: Em muitos casos, um algoritmo de busca aleatório será muito mais eficiente se comparado a um algoritmo determinístico, isto ocorre pois o tempo gasto em decidir a próxima estratégia de um algoritmo determinístico, no caso do algoritmo aleatório, é investido em explorar e avaliar uma quantidade maior do universo de possíveis soluções.
- Flexibilidade: Algoritmos de busca aleatórios possuem a flexibilidade de variar entre algoritmos puramente aleatórios e altamente determinísticos de acordo com o problema em análise e o estágio de busca.
- Informação provida e utilizada: Algoritmos de busca aleatória podem coletar informações sobre a função de busca em execução e utilizar esta informação para guiar a busca, apesar destar informações serem do tipo local, podem ser utilizadas para otimização.

Em contraponto à facilidade de construção proporcionada, o algoritmo aleatório possui a desvantagem do tamanho dos casos de testes gerados, este fator pode afetar a execução dos casos de teste dado que o tempo de execução e validação de cobertura destes aumentará (SHAMSHIRI *et al.*, ). O algoritmo também é dificultado por determinados tipos de entrada como constantes e valores específicos de strings, ambos necessitam algoritmos de *seeding* no auxílio da geração (SHAMSHIRI *et al.*, ).

## *3.3 Algoritmos de busca baseados em evolução*

Algoritmos evolutivos são inspirados na evolução natural, estes algoritmos têm se mostrado eficazes quando utilizados em diferentes tipos de problemas de otimização (CAMPOS *et al.*, 2017).

A população é gradualmente otimizada usando operações geneticamente inspiradas como a operação de *crossover*, esta operação combina material genético de no mínimo dois indivíduos para assim gerar uma nova prole. Outra operação é a de mutação, uma operação na qual altera-se material genético em um indivíduo a uma baixa probabilidade, para assim garantir diversificação genética (CAMPOS *et al.*, 2017). No contexto de geração de suítes de teste, os indivídios da população são caracterízados por conjuntos de casos de teste, também chamados de suítes de teste. (CAMPOS *et al.*, 2017)

A seleção de indivíduos é guiada por uma função de aptidão, a qual pontua melhor os indivídios mais adaptados para um problema de otimização (CAMPOS *et al.*, 2017). A função de aptidão baseia-se em critérios como cobertura de código a cobertura de ramificações de código (CAMPOS *et al.*, 2017), sendo assim indivíduos com valores altos de aptidão (os quais cobrem mais critérios) são mais adaptados ao problema em questão e mais prováveis de sobreviver a novas iterações e produzirem novos candidatos. (CAMPOS *et al.*, 2017)

É importante salientar que o aumento do número de objetivos de cobertura pode afetar o desempenho do algoritmo evolutivo (CAMPOS *et al.*, 2017)

### 3.3.1 Algoritmos genéticos

Os algoritmo genéticos estão entre os algoritmos evolutivos mais usados em diversos problemas de otimização pois podem ser facilmente implementados e apresentam bons resultados (CAMPOS *et al.*, 2017). No contexto de geração de suítes de testes e algoritmos genéticos, os indivídios da população são caracterízados por conjuntos de casos de teste, também chamados de suítes de teste (CAMPOS *et al.*, 2017). Os algoritmos genéticos geralmente possuem a seguinte estrutura lógica:

**Algoritmo 1** Algoritmo genético padrão com as seguintes entradas: Condição de parada $C$, função de aptidão $A$, função de seleção $Fs$, tamanho de população $Tp$, função de crossover $Fc$, probabilidade de crossover $Pc$, função de mutação $Fm$ e probabilidade de mutação $Pm$.

```
1:  P ← PopulacaoRandomicamenteGerada(ps)
2:  ValidarAptidão(ps)
3:  Enquanto ¬C faça
4:      Np ← []
5:      Enquanto |Np| < Tp faça
6:          p1, p2 ← Selecionar(Fs, P)
7:          o1, o2 ← Crossover(Fc, Pc, p1, p2)
8:          Mutacao(Fm, Pm, o1)
9:          Mutacao(Fm, Pm, o2)
10:         Np ← Np ∪ {o1, o2}
11:     Fim enquanto
12: P ← Np
13: ValidarAptidão(A,P)
14: Fim enquanto
15: retornar P
```

Fonte: (CAMPOS *et al.*, 2017)

No algoritmo 1 é possível verificar os passos de execução na versão padrão do algoritmo genético, o algoritmo começa gerando uma população aleatória de indivíduos os quais são potenciais soluções para o problema em análise (linha 1). Em seguida, enquanto a condição de parada $C$ não for atingida, os seguintes passos serão executados para cada indivíduo : seleção (linha 6), crossover (linha 7) e mutação (linhas 8 e 9). Por fim, a cada uma dessas iterações, a funcão de aptidão é aplicada (linha 13) com o intuito de pontuar indivíduos com a melhor aptidão para resultado na solução em questão para que estes não sejam descartados na função de seleção da próxima iteração linha(6).

Diversas variações do algoritmo genético padrão foram propostas com o intuito de melhorar sua efetividade (CAMPOS *et al.*, 2017). Uma das variações de algoritmo genético é sua versão monotônica do algoritmo genético padrão (*Standard GA*), a versão monotônica somente inclui, após operações de mutação e validação em cada iteração, somente o melhor conjunto da prole ou o melhor indivíduo da prole anterior (CAMPOS *et al.*, 2017). Dentre as variações de algoritmos genéticos padrão, existe a variação *Steady State GA,* esta variação utiliza a mesma estratégia de substituição presente na variação monotônica, a prole descendente substitui os ascendentes da população atual imediatamente a ocorrência da fase de mutação. (CAMPOS *et al.*, 2017)

O algoritmo evolutivo 1 + ($\lambda$,$\lambda$)

A versão do algoritmo evolutivo 1 + ($\lambda$,$\lambda$) , introduzida por Doerr, possui como primeiro passo a geração de uma população aleatória de tamanho 1, em seguida, a operação de mutação é aplicada para criar $\lambda$ versões diferentes com mutação do indivíduo em análise (CAMPOS *et al.*, 2017). A alta probabilidade de mutação ocorre para incentivar que a exploração seja mais rápida no espaço de busca disponível. (CAMPOS *et al.*, 2017)

No próximo passo, o operador de mutação é aplicado novamente com uma probabilidade maior de mutação, dada por K, onde K é usualmente maior que um, isto permite que, na maioria dos casos, mais de um gene seja mutado por cromossomo. Por fim, a operação de crossover acontece uniformemente aos ascendentes e aos melhores descendentes para criar uma prole de tamanho $\lambda$ (CAMPOS *et al.*, 2017). A operação de *crossover* unificada entre o melhor indivíduo com os lambda mutantes e seu ascendente foi proposta para amenizar os defeitos causados pela mutação agressiva, após essa operação, toda a prole e avalidada e somente o melhor indivíduo é selecionado (CAMPOS *et al.*, 2017)

Se o melhor descendente é melhor avaliado do que o seu ascendente, a população criada de tamanho um é então substituída pela melhor prole. (CAMPOS *et al.*, 2017)

O algoritmo evolutivo $\mu + \lambda$

O algoritmo evolutivo $\mu + \lambda$ é um algoritmo baseado em mutações em que o número de ascendentes e descendentes são restringidos respectivamente a $\mu$ e $\lambda$ (CAMPOS *et al.*,

2017). Cada gene sofre mutação independente com probabilidade 1/n, após a operação de mutação, a prole gerada é comparada com cada asendente com o intuito de manter os melhores ascendentes presentes até então (CAMPOS *et al.*, 2017).

O algoritmo evolutivo MOSA

O algoritmo MOSA, Many-Objective Sorting Algorithm, contrapõe os outros algoritmos por tratar cada objetivo de cobertura como um objetivo de otimização independente (CAMPOS *et al.*, 2017). Para que seja possível trabalhar com o grande número de objetivos de cobertura, o qual ocorre por conta da combinação dos diversos objetivos de cobertura, a extensão do algoritmo DynaMOSA dinamicamente seleciona os critérios (CAMPOS *et al.*, 2017).

Tanto o algoritmo MOSA quanto o algoritmo DynaMOSA demonstraram atingir alta cobertura de alguns critérios selecionados quando comparados aos algoritmos genéticos tradicionais. (CAMPOS *et al.*, 2017)

## *3.4 Algoritmo de busca aleatória ou algoritmos evolutivos ?*

Apesar de muitos dos aspectos destes algoritmos avaliados em detalhes, a influência de diferentes algoritmos em diferentes casos recebeu pouca atenção na literatura (CAMPOS *et al.*, 2017). Vários algoritmos foram propostos porém nenhum dos algoritmos pode ser considerado melhor em todos os domínios de problemas, alguns algoritmos desempenham melhor do que outros de acordo com as características presentes no domínio do problema. (CAMPOS *et al.*, 2017)

É teoricamente impossível desenvolver um algoritmo de geração de dados de teste o qual desempenha bem em todos os tipos de problemas disponíveis (CAMPOS *et al.*, 2017). Uma abordagem comum em problemas de geração de dados de testes na engenharia de software é utilizar um algoritmo genético primariamente e somente depois refiná-lo ou compará-lo com outros algoritmos para assim verificar qual dos algoritmos e soluções disponíveis é apropriado para o problema em questão. (CAMPOS *et al.*, 2017)

O estudo promovido por (CAMPOS *et al.*, 2017) demonstra que algoritmos genéticos claramente desempenharam melhor do que algoritmos aleatórios, em média os algoritmos

evolutivos possuem maior cobertura quando comparados a algoritmos de busca aleatória. (CAMPOS *et al.*, 2017)

Existe a necessidade compreender as diferentes características de software as quais influenciam diferentes coberturas de teste nas diferentes técnicas para que assim seja possível desenvolver uma hyper-heurística a qual seleciona a adapta o algoritmo para cada problema de geração em específico (CAMPOS *et al.*, 2017).

## *3.5 EvoSuite*

A EvoSuite é uma ferramente baseada em técnicas de busca que também utiliza algoritmos genéticos na geração automatica de suítes de testes para classes escritas em Java (FRASER *et al.*, 2017). Para uma determinada classe, referenciada na ferramenta como alvo, e seu *classpath* completo, o caminho o qual o ambiente de execução possa encontrar as classes compiladas como bytecode e respectivas dependências, a EvoSuite automaticamente produz um conjunto de testes unitários escritos para o framework de execução de testes Junit com o intuito de maximixar a cobertura de código. (FRASER *et al.*, 2017).

A efetividade da EvoSuite tem sido avaliada tanto como uma ferramenta de código aberto quanto uma ferramenta de apoio a testes nas áreas industriais se tratando de cobertura de código, efetividade na busca de falhas e inclusive aumentando a produtividade de desenvolvimento (FRASER *et al.*, 2017).

A ferramenta conseguiu o primeiro lugar nas segunda e quarta edições do *Unit testing tool competition* além de conseguir o segundo lugar na terceira edição (FRASER *et al.*, 2017), a configuração da EvoSuite para a competição em maioria manteve seus valores padrão, já que estes foram ajustados extensivamente (FRASER *et al.*, 2017). Com um *budget* de 480 segundos, a ferramenta atingiu em média 66.5% de cobertura de caminhos, 0.9% superior se comparado ao ano anterior com um score de mutação de 50.7%, 9.7% super do que o ano anterior. (FRASER *et al.*, 2017)

Além das últimas melhorias, diversas correções foram aplicadas desde a última edição do *Unit testing tool competition* em 2017 (FRASER *et al.*, 2017), no ano de 2017 a ferramenta passou a utilizar mocks em seus testes através da ferramenta Mockito [1] (FRASER *et al.*, 2017). Após atingir determinada porcentagem do tempo de busca

[1] https://site.mockito.org/

disponível, a ferramenta considera o uso de *mocks* ao invés de classes reais. (FRASER *et al.*, 2017)

# 4 Métricas de Softwares orientados a objetos

Projeto e desenvolvimento de softwares orientados a objeto são conceitos populares no cenário atual de desenvolvimento de software (SRIVASTAVA; KUMAR, 2013), neste contexto é notável que classes e métodos são estruturas básicas neste gênero de software (KAN, 1995).

A análise e projeto de softwares orientados a objeto focam nos objetos como estruturas primárias de computação (KAN, 1995) (SRIVASTAVA; KUMAR, 2013), nesta cada classe é composta por dados e operações realizadas, os quais são representações de uma única entidade real de negócio (SRIVASTAVA; KUMAR, 2013).

As operações realizadas são caracterizadas pelos métodos das classes, a quantidade de funcionalidade, representada por método, agregada ao um software orientado a objeto pode ser estimada com base na quantidade de classe, métodos e variantes destes, utilizados (KAN, 1995). Desta maneira, é comum relacionar métricas de softwares orientados a objeto a valores baseados em atributos de classes e métodos, sejam linhas de código, complexidade, entre outros (KAN, 1995).

As métricas de software tornaram-se elementos primordiais em diversos domínios de Engenharia de software, e principalmente no âmbito de garantia de qualidade, já que toda a informação reunida por métricas pode ser submetida a diversos tipos de análise e comparação com dados históricos com o objetivo de avaliar e garantir qualidade de software (SRIVASTAVA; KUMAR, 2013). Métricas de software podem ser utilizadas para predizer atributos de qualidade em tempo de execução, inclusive para aplicações cujo requisito é ser uma aplicação de tempo real (SRIVASTAVA; KUMAR, 2013). O real valor de métricas de software vem da sua representação dos atributos de software externos, muitos destes com alto valor de negócio, como confiabilidade, mantenibilidade, reusabilidade, testabilidade e eficiência (SRIVASTAVA; KUMAR, 2013), os quais são requisitos de qualidade descritos pela ISO9126 (ZEISS *et al.*, 2007).

## *4.1 Medidas diretas e indiretas*

Uma medida direta de software é uma métrica cujo cálculo de seu valor não depende nenhuma outra métrica (SRIVASTAVA; KUMAR, 2013). Exemplos claros de medidas

em produtos são: LOC (*Lines of code*), velocidade de execução, tamanho de memória, quantidade de defeitos reportados (SRIVASTAVA; KUMAR, 2013).

Uma medida indireta de software é uma métrica que envolve a medida de um ou mais atributos de software(SRIVASTAVA; KUMAR, 2013). As medidas diretas são, em geral, mais fáceis de serem coletadas.

## *4.2 O conjunto de métricas CK*

Em 1994 Chidamber e Kemerer propuseram seis métricas de projeto e complexidade de softwares orientados a objeto, as quais, futuramente, tornaram-se o conjunto de métricas CK (KAN, 1995). Este conjunto foi derivado aplicando

Chidamber e Kemerer aplicaram as seis métricas do conjunto (as quais serão definidas nas próximas seções) em estudos empíricos de duas empresas, uma usando C++ e outra Smalltalk (KAN, 1995), o sumário desta é demonstrado na tabela.

### 4.2.1 *Weighted methods per class (WMC)*

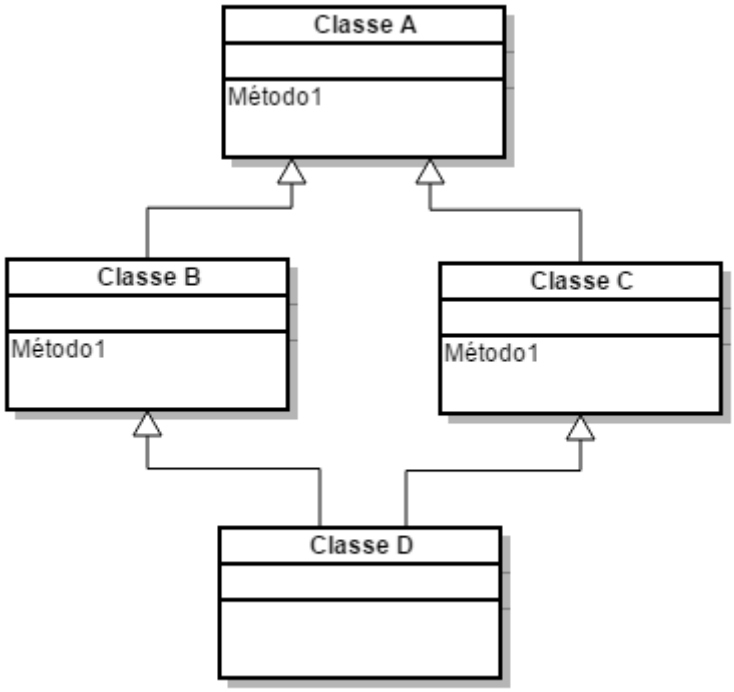
A WMC é uma métrica contabilizada pela soma das complexidades dos métodos, cuja complexidade é dada pelo *cyclomatic complexity number* de McCabe (CCN)(KAN, 1995) (WATSON; MCCABE; WALLACE, 1996). Medir de maneira automatizada o *cyclomatic complexity number* não é trivial de implementação dado que nem todos os métodos são acessíveis de acordo com a hierarquia dos objetos, isso é resultado por conta da herança aplicada no projeto do componente (KAN, 1995).

### 4.2.2 *Depth of inheritance tree (DIT)*

A DIT é uma medida que compreende o valor do comprimento máximo de um caminho em uma árvore de herança, este comprimento é dado pela distância entre o nó em análise até o nó raíz (KAN, 1995). A definição da DIT baseia-se na premissa de que desenvolvedores lidam com linguagens de programação orientadas a objeto as quais permitam que uma classe possua no máximo uma classe pai (BRUNTINK; DEURSEN, 2004), linguagens alheias a este conceito são conhecidas por possuir herança múltipla,

presente em linguagens como C++ e Python. Muitas linguagens orientadas a objeto não suportam herança múltipla por conta do problema do diamante, caracterizado pela seguinte situação:

Figura 4 – Exemplo de problema do diamante




Fonte: Gustavo Ramos, 2017

No caso do problema do diamante, um objeto instanciado do tipo D possui acesso para executar o **Método 1**, porém o compilador pode não saber a qual referência de **Método 1** executar já que o objeto do tipo D herda tanto de B e de C quanto de A, todos os quais possuem o **Método 1**. Este caso é conhecido por problema do diamante, por conta deste, linguagens como Java não permitem a herança múltipla.Dada a premissa de que a linguagem do software em análise não suporta herança múltipla, ao calcular o DIT, o número de ancestrais do nó $c$ em análise, corresponde à profundidade de $c$ na árvore de herança (BRUNTINK; DEURSEN, 2004). A fórmula que descreve o cálculo da DIT para um determinado nó $c$ é dado a seguir:

$$DIT(c) = |Ancestrais(c)| \tag{1}$$

### 4.2.3 *Number of children of a class (NOC)*

O NOC é uma métrica de simples cálculo, caracterizada pelo número de sucessores imediatos(subclasses) na árvore de hierarquia, calculada com base em uma classe $c$ em análise (KAN, 1995). Seu cálculo é dado pela fórmula a seguir:

$$NOC(c) = |QuantidadeDeFilhos(c)| \tag{2}$$

### 4.2.4 *Coupling between object classes (CBO)*

Uma classe $A$ é acoplada a uma classe $B$ se a classe $A$ invoca uma método ou utiliza uma variável de uma instância de $B$ (KAN, 1995), sendo assim, o CBO é dado pelo número de classes a qual uma classe $c$ em análise está acoplada, ou seja, seu cálculo é dado por:

$$CBO(c) = |Quantidadedeclassesquerelaciona(c)| \tag{3}$$

### 4.2.5 *Response for a class (RFC)*

O RFC é caracterizado pelo número de métodos que podem ser executados na resposta de uma mensagem recebida por uma instância de uma classe (KAN, 1995), ou seja, o RFC é uma contagem do número de métodos de uma classe $c$ em análise e o número de métodos de outras classes que são invocados pelos métodos de $c$ (BRUNTINK; DEURSEN, 2004).

Quanto maior o número de métodos que podem ser invocados indiretamente através de uma chamada de um método, maior a complexidade de uma classe, basicamente o RFC captura o tamanho de conjunto de respostas de uma classe (KAN, 1995). O RFC é calculado pelo número de métodos locais mais o número de métodos chamados por métodos locais (KAN, 1995).

### 4.2.6 *Lack Of Cohesion Of Methods (LCOM)*

A independência pode ser medida por critérios qualitativos: coesão e acoplamento, a coesão é a medida relação funcional de um módulo (PRESSMAN, 2009), neste contexto, a coesão de uma classe é dada pela proximidade de métodos locais à instâncias de variáveis na classe, alta coesão indica uma boa subdivisão das classes (KAN, 1995).

A métrica LCOM representa a dissimilaridade dos métodos em uma classe pelo uso de instâncias de variáveis, a baixa de coesão aumenta a complexidade e por consequência, oportunidade de ocorrência de erros durante o processo de desenvolvimento (KAN, 1995).

## *4.3 Modelo proposto por Srivastava e Kumar*

Srivastava e Kumar proporam em 2013 um modelo de métricas para validação de qualidade de software com a justificativa de que nenhuma das métricas tratavam acoplamento ou similaridades como relações transacionais, sendo assim, métricas que mapeiam estas características são passíveis de incorporação no modelo. No estudo em questão, diversos dados foram coletados de projetos orientados a objeto nas linguagens Java e C++, no estudo em questão foi constatado que diversas métricas são baseadas em ideias similares e representam informações redundantes, sendo assim, foi validado que um subconjunto de métricas pode ser utilizado para predição de falhas.

O modelo proposto por (SRIVASTAVA; KUMAR, 2013) atingiu um nível de acurácia superior a 80% na predição de falhas em classes. É observado na literatura que métricas de softwares orientados a objetos aliadas aos modelos de classificação e predição existentes possuíram efetividade em encontrar relação entre identificação e classificação de falhas e qualitdade de software (SURESH, 2016).

O fato ocorre pois métricas de software são a melhor ferramenta para avaliar a qualidade de softwares orientados a objetos (SURESH, 2016). É observado que muitos conjuntos de métricas disponíveis provém informações de qualidade de softwares orientados a objetos. (SURESH, 2016)

## *4.4 Outras métricas estáticas*

Além de características que possam ser descritas por métricas C.K, é possível também utlizar métricas que descrevam características de métodos, já que métodos compõe as ações presentes dem classes. No traballho desenvolvido por (ELER; ENDO; DURELLI, 2016), diversas métricas de métodos foram extraídas utilizando execução simbólica, métricas como: Número de loops (NLp)número de loops aninhados (NNLp); quantidade de constantes (NCtt); número de variáveis (NVar); quantidade de variáveis do tipo inteiro (NInt); quantidade de variáveis do ponto flutuante (NFit); quantidade de referências nulas (NNull); Quantidade de variáveis do tipo string (NStr); número de objetos (NObj); Número de arrays (NArr).

É possível também extrair métricas como número de métodos (Nom) , númeto de métodos públicos (Nopm), número de métodos estáticos (Nosm), número de atributos (Nof), número de atributos públicos (Nopf), número de atributos estáticos (Nosf), número de invocação de métodos estáticos (Nosi) e total de linhas de código (Loc).

# 5 Mineração de dados

Mineração de dados é o processo de extrair conhecimento valioso de um grande agrupamento de dados, desconhecido e de valor para o problema em questão (SETH; BANKA, 2016; SHARMA; SINGH; SINGH, 2015). Na mineração ocorre a análise de dados a partir de diferentes perspectivas com o intuito de obter informaçãos úteis para tomada de decisão (SHARMA; SINGH; SINGH, 2015). Ela é utilizada em diversos campos como robótica, decisões de marketing, decisões corporativas, inteligência de negócio e segurança (SETH; BANKA, 2016).

Na área de mineração de dados, distinguir diferentes classes a partir de um volume de dados e classificar a partir destes é uma técnica importante (SETH; BANKA, 2016), sendo assim, a classificação é uma subárea importate de mineração de dados, dado que ela é capaz de analisar dados e organizá-los em classes a partir de suas características (SETH; BANKA, 2016).

## *5.1 Classificação*

A classificação é uma técnica vastamente utilizada no domínio de mineração de dados, na qual escalabilidade e eficiência são requisitos imediatos em classificação de grandes conjuntos de dados (SHARMA; SINGH; SINGH, 2015). A classificação é uma atividade não supervisionada (SHARMA; SINGH; SINGH, 2015), a qual já existe o conhecimento desde o início quais são as classes disponíveis a serem utilizadas pelo algoritmo, o algoritmo então usa a informação de classes e características presentes em um conjunto de dados para treinamento e validação (SHARMA; SINGH; SINGH, 2015).

Um problema de classificação é conhecido estruturalmente por receber uma entrada de dados, conhecido por conjunto de dados de aprendizagem (dataset), o qual é caracterizado por uma série de características (features) de análise e respectivas classes (SETH; BANKA, 2016).

O principal objetivo de algoritmos de classificação é construir um modelo de classificação utilizando o conhecimento obtido a partir do conjunto de dados de aprendizagem e avaliar uma porcentagem desdes a partir do conhecimento adquirido (SETH; BANKA, 2016).

Diferentes algoritmos de classificação usam diferentes técnicas para encontrar relação entre os dados e as classes (SHARMA; SINGH; SINGH, 2015), estas relações estão sumarizadas em um modelo, o qual pode ser aplicado em novas instâncias para classificação destas (SHARMA; SINGH; SINGH, 2015).

Um modelo pode também ser utilizado para validação, em que novas instâncias são submetidas a um modelo e avaliadas em seguida para verificar se foram classificadas corretamente, este processo é chamado de validação do modelo (SHARMA; SINGH; SINGH, 2015).

Dentre os classificadores mais famosos, o mais conhecido é o classificador bayseano. (SETH; BANKA, 2016), o classificador bayseano é um classificador construído com base no teorema de bayes (SETH; BANKA, 2016). Quando aplicado a grandes conjuntos de dados, classificadores bayseanos conseguem alta acurácia em alta velocidade (SETH; BANKA, 2016), a forma mais simples de classificador bayseano conhecido atualmente é o classificador *naive bayes* (SETH; BANKA, 2016).

## *5.2 Classificador naive bayes*

O teorema bayseano deriva da estatística bayseana, o qual possui premissas fortes e ingênuas (SURESH, 2016). O classificador bayseano é uma abordagem probabilística simples o qual assume que todos os atributos de um tupla pertencem a uma dada classe com uma probabilidade independente de ser classificado em relação a outros atributos (SURESH, 2016).

O teorema bayseano é uma afirmação estatística que permite o cálculo de probabilidades condicionais, probabilidades condicionais são probabilidades que refletem a influência de um evento na probabilidade de outro evento (SHARMA; SINGH; SINGH, 2015) .

Os termos geralmente utilizados em estatística bayseana são: probabilidade anterior e posterior: A probabilidade anterior é o evento com sua probabilidade original, antes de obter qualquer informação adicional em contraponto, a probabilidade posterior é a probabilidade anterior seguida de alguma informação adicional (SHARMA; SINGH; SINGH, 2015). O teorema de bayes pode ser descrito como (SHARMA; SINGH; SINGH, 2015):

$$P(A|B) = \frac{P(B|A).P(A)}{P(B)} \tag{4}$$

Onde:

$P(A)$ é a probabilidade anterior de A

$P(B)$ é a probabilidade anterior de B

$P(A|B)$ é a probabilidade posterior de A dado B

$P(B|A)$ é a probabilidade posterior de B dado A

O classificador naive bayes funciona da seguinte maneira (SETH; BANKA, 2016):

1. Inicialmente é necessário um conjunto de treinamento $C$ com $k$ caraterísticas e $M$ classes associadas, usando o valor de atributos presente em cada tupla do dataset, caracterizado por um vetor $\alpha$=($\alpha$1, $\alpha$2, $\alpha$3,. . . ., $\alpha$k);
2. A partir de um conjunto um conjunto de $M$ classes beta compostos por $\gamma$1, $\gamma$2, $\gamma$3,. . . ., $\gamma$m; O papel do classificador é predizer a partir de uma tupla $\alpha$ qual a classe $\gamma$ com maior probabilidade a qual esta tupla pertence dado o que foi aprendido com o conjunto de treinamento, ou seja, o classificador prediz se uma tupla $\alpha$i pertence a uma classe $\gamma$i a partir da seguinte fórmula:

$$P(yi|alpha) > P(yj|alpha), \text{ para todo } 1 \leq \text{j} \leq \text{m, j} \neq \text{i} \tag{5}$$

Consequentemente, P($\gamma$i—$\alpha$) deve ser maximizado, a classe resultante com o maior valor de P($\gamma$i—$\alpha$) é chamada de probabilidade posterior, como no teorema de *bayes*. Dado o fato de que para todas as classes P($\alpha$) é constante, maximizar P($\alpha$—yi) * P(yi) garante o propósito (SHARMA; SINGH; SINGH, 2015).

Um conjunto de treinamento com muitos atributos para o cálculo de P($\alpha$) pode ser computacionalmente custoso de ser calculado, sendo assim, o classificador reduz este custo assumindo que as classes são condicionalmente independentes entre si (SHARMA; SINGH; SINGH, 2015).

## *5.3 Weka*

A ferramenta Weka (*Waikato Environment for Knowledge Learning*)[1] é um software desenvolvido pela universidade de Waikato na Nova Zelândia (SUBBULAKSHMI; DEEPA; MALATHI, 2012). A WEKA apoia diferentes tipos de atividades como pré-processamento, classificação, clusterização, regressão, visualização e seleção de atributos (SUBBULAKSHMI; DEEPA; MALATHI, 2012). A premissa da aplicação é utilizar-se de um computador e o software para executar técnicas de aprendizado de máquina e mineração de dados e derivar informações úteis através de padrões (SUBBULAKSHMI; DEEPA; MALATHI, 2012).

A feramenta possui em torno de duas décacas, com implementações dos algoritmos de classificação e interfaces gráficas de fácil uso e visualização além de algoritmos de validação (BOUCKAERT *et al.*, 2008).

A Weka opera com dados unidimensionais, permitindo que pessoas de qualquer nível de conhecimento em mineração de dados possam identificar informações escondidas em seus conjuntos de treinamentos com uma interface simples e direta (SUBBULAKSHMI; DEEPA; MALATHI, 2012). Após carregar e treinar um modelo com um conjunto de dados, o classificador pode ser validado para o conjunto e as classes fornecidas (SUBBULAKSHMI; DEEPA; MALATHI, 2012).

[1] https://www.cs.waikato.ac.nz/ml/weka/

# 6 Materiais e métodos

Neste capítulo serão apresentados os materiais utilizados e a maneira como estes materiais foram manipulados por meio dos métodos também descritos neste capítulo. Para efeito de organização, este capítulo será dividido em seções que explicam desde os materiais e métodos utilizados, descrevendo os passos separadamente.

## *6.1 Materiais*

Durante o processo de pesquisa, foi utilizado um conjunto de 110 projetos os quais são reconhecidos e utilizados por diversos pesquisadores da área. Os projetos em questão fazem parte do corpus SF110, o qual é caracterizado por um conjunto de 110 projetos open source retirados do site SourceForge[1] compostos por: 100 projetos aleatoriamente escolhidos somados a 10 dos projetos com mais download em junho de 2014. (SHAMSHIRI *et al.*, ).

O corpus SF110 foi utilizado como objeto de análise pelo processo de pesquisa por conta de sua diversidade de características, isso ocorre pois os projetos que compõe o corpus resolvem problemas presentes em domínios variados. A partir do corpus, diversas ferramentas foram utilizadas para extrair informações e processá-las, o uso destas será detalhada na seção de métodos, para este processos as seguintes ferramentas foram utilizadas

- **EvoSuite[2]:** ferramenta utilizada para geração dos dados de teste nas duas técnicas apresentadas no capítulo 3 e cálculo da porcentagem de cobertura de código atingida;
- **CK[3]:** ferramenta disponível em repositório opensource para extração de métricas CK em projetos Java;
- **Weka[4]:** ferramenta de mineração de dados utilizada para classificação pela técnica citada no capítulo 5;
- **LibreOffice Calc[5]:** ferramenta utiizada para o cálculo das correlações entre os valores que serão apresentados.

[1] https://sourceforge.net/
[2] http://www.evosuite.org/
[3] https://github.com/mauricioaniche/ck
[4] https://www.cs.waikato.ac.nz/ml/weka/
[5] https://www.libreoffice.org/discover/calc/

- **Go** [6]**:** linguagem de programação utilizada para codificação de algumas rotinas de processamento dos dados.
- **Maven** [7]**:** ferramenta compilação, gestão de dependências e empacotamento dos projetos java presente no SF110.

Nas seções posteriores será descrita a metodologia aplicada durante o processo de pesquisa e os resultados encontrados, o processo foi segmentado nas tarefas de extração e análise das métricas, geração dos dados de teste, associação dos resultados, mineração dos dados, teste da hipótese e análise dos resultados.

## *6.2 Métodos*

Esta seção visa descrever a metodologia utilizada durante esta dissertação, detalhando as etapas necessárias para conclusão de todas as atividades de análise, condução dos experimentos e resultados. As etapas as quais compõem a pesquisa conduzida durante esta dissertação podem ser divididas em :

- Análise bibliográfica de métricas em softwares orientados a objetos;
- Estudo dos algoritmos de geração automática de dados de teste e comparativos entre os algoritmos utilizados;
- Escolha, análise e preparação da base de dados;
- Escolha das ferramentas de coleta de métricas, geração de dados de teste e mineração de dados;
- Geração dos dados de teste e coleta dos medidas apresentadas pelos dados de saída;
- Extração das métricas selecionadas;
- Consolidação das métricas de geração de dados de teste e das métricas de softwares orientados a objetos;
- Análise dos dados consolidados e montagem dos conjuntos de treinamento;
- Execução do algoritmo de classificação;
- Análise dos resultados.

[6] https://golang.org/
[7] https://maven.apache.org/

### 6.2.1 Análise bibliográfica de métricas em softwares orientados a objetos

O conjunto de métricas utilizado é o conjunto proposto por Chidamber & Kemerer, durante a análise bibliográfica, diversas métricas foram estudadas para que assim fossem escolhidas, entre o conjunto de métricas existente, as métricas que melhor representassem características de softwares orientados a objetos. Além do requisito claro de representação de características do software, as métricas necessitam ser de fácil e rápida extração, uma vez que softwares que necessitem escolher uma técnica mais adequada necessitam que a escolha ocorra em tempo hábil.

As métricas de softwares orientados a objetos podem ser divididas em métricas estáticas e métricas dinâmicas, sendo que o primeiro conjunto não necessita de execução do $SUT$ para a extração diferentemente do segundo que demanda que as métricas sejam extraídas em tempo de execução. Por conta destas características, as métricas estáticas são extraídas mais rapidamente e consomem menos recursos computacionais, sendo assim, para preencher os requisitos de tempo e processamento, visto que a seleção do melhor algoritmo necessita ser rápida, as métricas estáticas foram escolhidas.

Dentre as métricas estáticas, vários tipos de medidas podem ser extraídas por meio de análise dos códigos das classes, por meio da análise bibliográfica foi possível notar que as métricas CK possuem boa representação das características de classes que compõe um software. O conjunto de métricas CK foi utilizado em diversos contextos como detecção de falhas e análise de qualidade dando suporte para descrição de características.

### 6.2.2 Estudo dos algoritmos de geração de dados de teste

A literatura sobre geração automática de dados de testes engloba diversas técnicas de busca que variam de técnicas aleatórias mais simples a técnicas baseadas em algoritmos genéticos e suas variações. Apesar do número de algoritmos disponíveis para uso, o objetivo de pesquisa deste trabalho baseia-se na escolha automática entre dois algoritmos distintos: os algoritmos de geração baseados em busca aleatória e o algoritmo de busca com base em algoritmos genéticos.

### 6.2.3 Definição da amostra utilizada

O experimento para validação das hipóteses possui diversas atividades de geração e extração de dados que serão detalhados nas próximas seções. Para que as extrações ocorram é necessária a utilização de uma base de amostras capaz de denotar os mais diversos tipos de classes presentes em softwares, ou seja, os tipos de problemas resolvidos pelos softwares que compõe a amostra devem ser dos mais variados possível.

Durante a análise dos algoritmos de geração de dados de teste foi possível notar o uso do SF110, também chamado de SF110 corpus, por diversos experimentos. O corpus SF110 possui grande relevância na literatura, sendo utilizado em experimentos relacionados à geração de dados de teste, além de ser um corpus consolidado, possui 110 projetos de software presentes em cenários variados de utilização, compondo em torno de 10.000 classes em Java, suprindo assim a necessidade de variedade de contextos de resolução de problemas.

Todos os arquivos escritos em Java que compõem os softwares presentes no SF110 podem ser utilizados a partir da coleta de métricas no experimento, com exceção de elementos como interfaces, sendo assim, a de análise ocorre para as classes definidas nos projetos. O mesmo ocorre para a geração de dados de teste, somente as classes de software em Java serão consideradas para análise.

### 6.2.4 Geração dos dados de teste

A geração dos dados de teste demanda uma ferramenta que seja capaz de gerar os testes de unidade nos dois algoritmos, além de fornecer estatísticas de geração como cobertura de código e tempo de execução.

Existem várias ferramentas de geração de dados de testes com base em buscas disponíveis em uma gama de linguagens de programação, principalmente Java e .NET. Ferramentas as quais variam de técnicas de random search como Randoop, Jcrasher, JTExpert, NightHawk, T3 ou Yeti-Test até ferramentas baseadas em técnicas evolutivas como EvoSuite, eToc ou Testful (SHAMSHIRI *et al.*, ).

Entre as ferramentas citadas, destaca-se a EvoSuite, a ferramenta possui mais de 53 publicações registradas no site oficial, foi submetida ao *Unit testing tool competition* e conseguiu o primeiro lugar em duas edições.

A EvoSuite suporta diversos algoritmos de geração como os citados no capítulo 3, dentre os algoritmos é possível escolher diferentes configurações como tempo máximo de uso em busca, técnicas de *seeding* para guiar busca, entre outras. As versões dos algoritmos utilizadas no *Unit testing tool competition* são as versões padrão dos algoritmos, utilizando também as configurações padrão de geração (SHAMSHIRI *et al.*, ).

Dado que o objetivo desta dissertação também consiste em analisar as diferenças dos algoritmos aleatório e genético e a desempenho da configuração adotada no *Unit testing tool competition* foi alta, atingindo mais de 80% de cobertura de código, então a versão monotônica padrão do algoritmo genético e a versão do algoritmo aleatório padrão são adotadas para o experimento, ambas com suas configurações padrão.

A execução do EvoSuite possui como resultado de saída um arquivo no formato CSV com os seguintes campos como cabeçalho: TARGET_CLASS, criterion, Coverage, Total_Goals, Covered_Goals.

A coluna TARGET_CLASS caracteriza o nome da classe e pacote da classe java, ela pode ser usada para unir os dados de métricas CK extraídos com os dados de cobertura, isso ocorre pois este é um dado disponível em ambas saídas das ferramentas; a coluna de nome Covered_Goals possui o percentual de critérios cobertos durante a execução. É importante salientar que o critério de cobertura adotado foi o de cobertura de caminhos.

### 6.2.5 Extração das métricas CK

As métricas de software são medidas extraídas a partir de componentes de software, as quais podem descrever características internas, inclusive atributos de qualidade dos componentes. O presente trabalho procura utilizar-se destes atributos, caracterizados por métricas, na seleção da técnica mais adequada de geração para cada classe presente em um software orientado a objetos.

A ferramenta selecionada para extração das métricas é a ferramenta CK publicada em repositório de código aberto Github, já descrita na seção de materias. A partir da

ferramenta é possível extrair a partir de um conjunto de classes escritas na linguagem de programação Java as seguintes métricas estáticas:

Tabela 1 – Métricas extraídas pelo CK

| Métrica | Descrição |
|---|---|
| CBO | Acoplamento entre os objetos |
| WMC | Complexidade de McCabe |
| DIT | Profundidade na árvore de herança |
| NOC | Número de classes as quais herdam a classe em análise |
| RFC | Número de invocações únicas de um método em uma classe |
| LCOM | Falta de coesão nos métodos |
| NOM | Número de métodos |
| NOPM | Número de métodos públicos |
| NOSM | Número de métodos estáticos |
| NOF | Número de atributos |
| NOPF | Número de atributos públicos |
| NOSF | Número de atributos estáticos |
| NOSI | Numero de invocação de métodos estáticos |
| LOC | Total de linhas de código |

Fonte: Gustavo da Mota Ramos, 2018

A tabela 1 lista todas as métricas extraídas pelo CK e a descrição do valor extraído em cada métrica. O CK foi escolhido por ser uma ferramenta que é não só capaz de extrair as métricas propostas por Chidamber & Kemerer, também é capaz de extrair outras métricas importantes em softwares orientados a objetos como as métricas de contagem de métodos, métricas de contragem de atributos e suas variações, ambas presentes na tabela 1.

A execução do CK possui como resultado de saída um arquivo no formato CSV com os seguintes campos como cabeçalho: file, class, type, cbo, wmc, dit, noc, rfc, lcom, nom, nopm, nosm, nof, nopf, nosf, nosi, loc. Além dos valores das métricas, a coluna class é importante para união dos dados de geração com suas respectivas métricas, dado que este é um campo presente em ambos arquivos de saída.

### 6.2.6 Consolidação dos dados extraídos e análises iniciais

Com a conclusão das atividades de extração das métricas e estatísticas de geração dos dados de teste, é necessário que os dados coletados de ambos sejam consolidados em um arquivo único de dados onde cada linha do arquivo possua a classe do software, as

métricas coletadas desta classe, o tempo necessário e a cobertura atingida por cada um dos dois algoritmos de geração de dados de teste. O arquivo final possui o nome de conjunto de dados consolidado.

O arquivo consolidado possui todos os dados de métricas e geração associados a casa uma das classes do SF110, isso permite que diferentes conjuntos de treinamento sejam criados com base em diferentes necessidades como, classificar pela técnica mais rápida ou pela técnica com maior cobertura.Com a montagem do conjunto de dados consolidado é possível conduzir algumas análises primárias.

### 6.2.7 Construção dos conjuntos de treinamento e análise dos resultados

A montagem do arquivo consolidado permite que estes diferentes conjuntos de treinamento sejam montados. Compreende-se por montagem do conjunto de treinamento, replicar os dados do arquivo consolidado em um novo arquivo e assim aplicar um campo a mais, chamado de classe.

Uma classe é simplesmente um valor de texto, no caso deste experimento ela determina o algoritmo mais adequado e aplicado no conjunto, este passo é importante para que o algoritmo de classificação saiba o conjunto de classes disponíveis no total e aprenda a classificar de acordo com as medidas fornecidas e as classes. As classes adotadas utilizadas então são **GA** para algoritmo genético e **RANDOM** para algoritmo aleatório.

Os experimentos de classificação demandam que os conjuntos de treinamento sejam montados de acordo com os diferentes critérios de análise nos experimentos. As análises desejadas por esta dissertação compreendem dois tipos de seleção do algoritmo de geração: o primeiro com base na maior cobertura de código e o segundo híbrido com base tanto na cobertura de código quanto no tempo geração.

O primeiro conjunto de treinamento é um conjunto baseado totalmente em cobertura de código, ou seja, selecionar o algoritmo de geração de dados de teste que atingiu a maior cobertura de código para cada tupla de uma classe java no arquivo consolidado. Sendo assim, para a montagem é necessário replicar os dados presentes no arquivo consolidado e aplicar nesta réplica a classe do algoritmo com maior cobertura, para cada linha: Aplicar **GA** se o algoritmo genético possuiu maior cobertura ou **RANDOM** se o algoritmo aleatório atingiu maior cobertura.

O segundo conjunto de treinamento é um conjunto baseado tanto na cobertura dos dados de teste quanto no tempo de execução. A aplicação da classe é diferente do conjunto anterior, quando a diferença de cobertura entre os algoritmos situa-se entre um percentual aceitável, a classe aplicada será a classe do algoritmo com o menor tempo. Assim como o conjunto de treinamento anterior, os dados consolidados necessitam ser replicados em um novo conjunto e as linhas iteradas para que a classe correta seja aplicada.

Supondo um valor como 5% como limiar, se a diferença de cobertura entre os dois algoritmos for menor ou igual a 5%, o algoritmo que consumiu menos tempo na geração de dados de teste será escolhido. O valor escolhido como limiar depende das análises primárias já descritas, espera-se com esse conjunto garantir maior velocidade com detrimento muito reduzido e aceitável da cobertura dos testes.

### 6.2.8 Classificação e análise dos resultados

A construção dos conjuntos de treinamento permite que o algoritmo de classificação seja executado e assim obter os dados para análise das hipóteses. Dentre os algoritmos de classificação disponíveis, o classificador *naive bayes* foi escolhido por apresentar boa performance de execução e bons resultados.

A velocidade de um classificador nestes experimentos é um fator importante, em um caso de implementação do classificador em uma ferramenta como a EvoSuite, o algoritmo precisa ser executado para cada classe do software, com o intuito de escolher o algoritmo mais adequado. Esta operação não pode ser de alto custo computacional ou demandar um tempo de execução que atrapalhe o processo.

A análise dos resultados apresentados pelos dados de saída baseia-se na assertividade de escolha apresentada no modelo proposto, ou seja, a quantidade corretamente classificadas de cada algoritmo. Este dado compõe o cálculo de precisão, a precisão é dada pelo proporção de instâncias corretamente classificadas pelo total de instâncias existentes (KAUR; KAUR; PATHAK, 2014).

Segundo (KAUR; KAUR; PATHAK, 2014), um resultado de classificação é considerado preciso se uma medida de performance como a precisão possui um valor maior ou igual a 70%. Espera-se com os modelos propostos atingir acima de 70% de instâncias corretas de classificação.

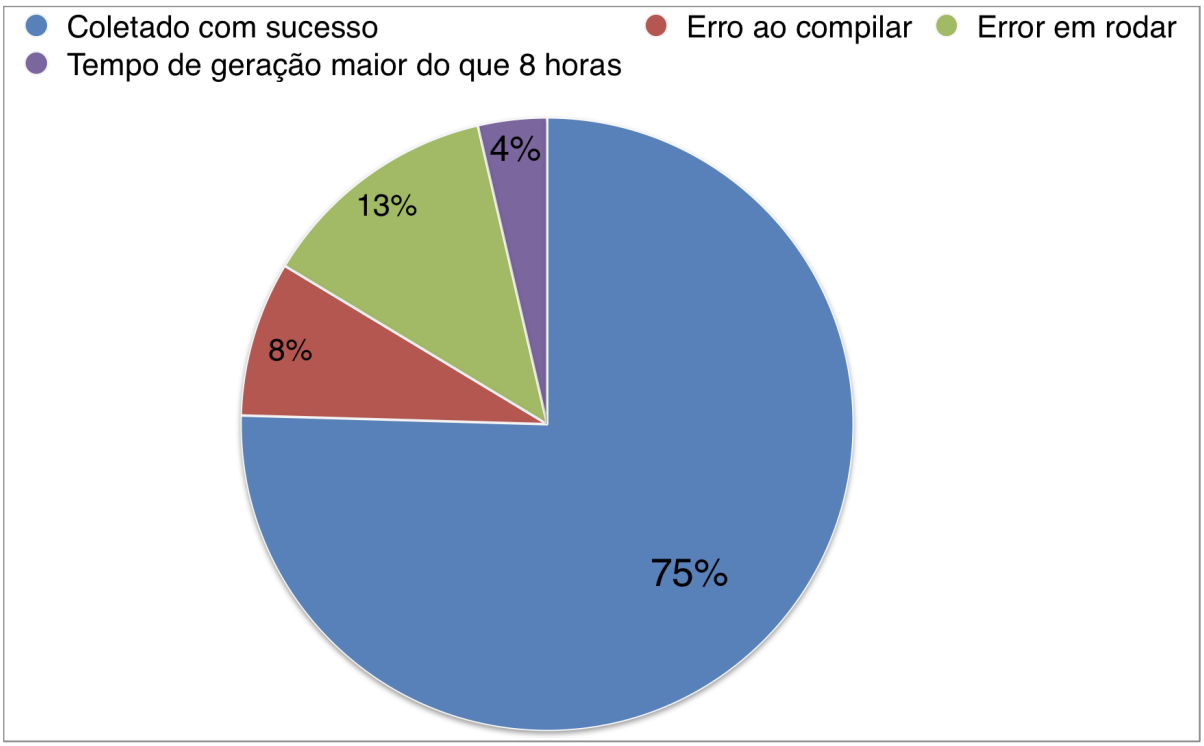
# 7 Condução dos experimentos e discussões

A primeira etapa do processo consistiu na extração dos dados para análises primárias das hipóteses apresentadas, este passo foi necessário pois a primeira análise estatística para validação foi a análise da correlação das métricas CK dos projetos e os dados de execução fornecidos da EvoSuite: Cobertura de código e tempo de execução. A análise da correlação é uma etapa de validação importante visto que valores de correlação com pouca ou nenhuma significância podem dar indícios da eficácia de seu uso na mineração de dados, esta etapa caracteriza como uma etapa eliminatória das etapas posteriores do processo.

Os dados necessários para a análise da correlação são as devidas quantidades de cobertura, tempo disponibilizados pela EvoSuite e métricas CK extraídas das classes do SF110. Por conta da velocidade de extrair métricas estáticas como as métricas CK, a primeira coleta ocorreu com a geração dos dados de teste pela EvoSuite, inicialmente com o algoritmo de busca genético padrão e em seguida com o algoritmo de busca aleatória.

O SF110 possui em torno de 10.000 classes java, espera-se então que o conjunto de dados consolidado final para treinamento possua o mesmo número, porém alguns dos projetos apresentaram problemas ao serem compilados , outros apresentaram mensagens de erro ao iniciar a geração de dados de testes utilizando algoritmos genéticos por meio da EvoSuite e alguns projetos possuíram tempo de execução superior a 8 horas.

Figura 5 – Distribuição de estado dos projetos após geração de dados de teste

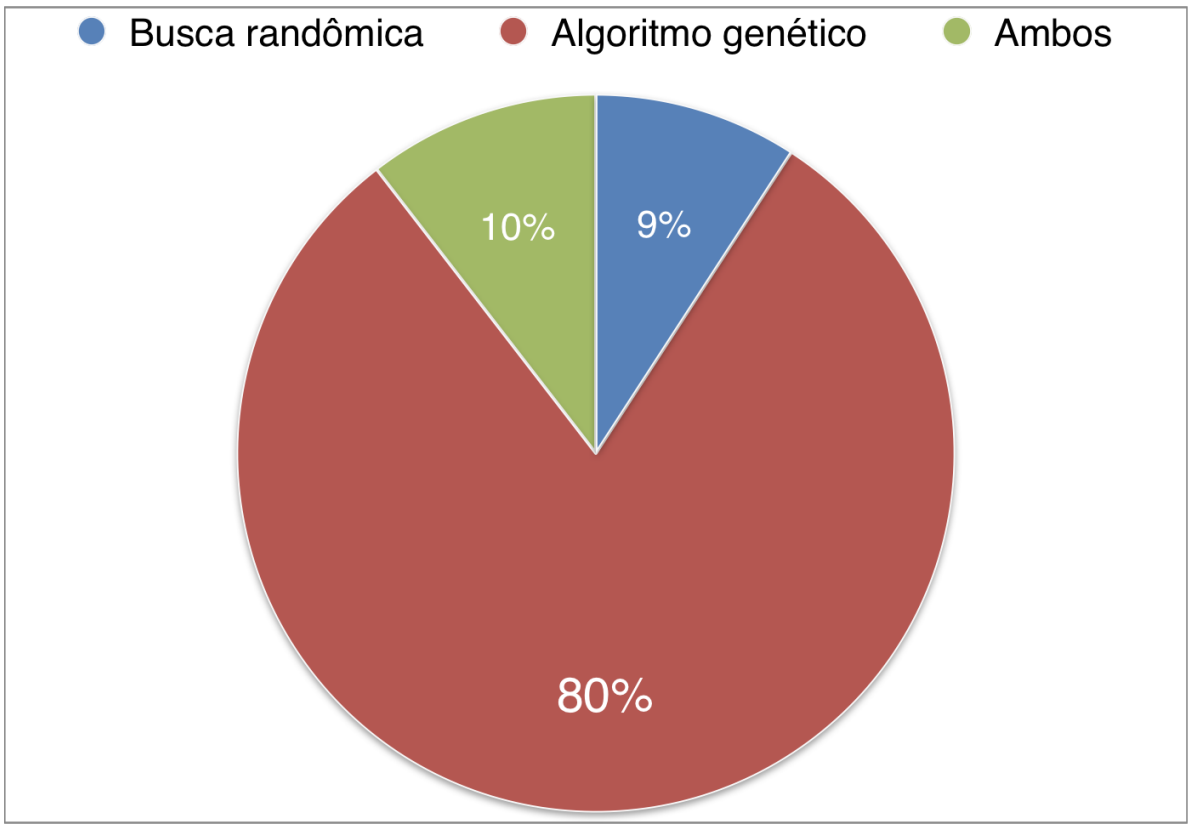



Fonte: Gustavo da Mota Ramos, 2018

Como demonstrado na figura 5, da quantidade total de projetos presentes no SF110, 75% tiveram seus dados de teste gerados corretamente, enquanto 8% apresentaram problemas ao serem compilados, 13% foram compilados corretamente porém apresentaram erros duranta geração dos dados de teste e 4% dos projetos precisaram ter sua execução interrompida pois o processo já havia atingido 8 horas de duração para um único projeto.

Seguido à geração de dados de teste por meio de algoritmos genéticos, a geração de dados de teste por meio da técnica aleatória ocorreu, porém esta ocorreu somente nos projetos com sucesso na geração anterior, isso ocorre por conta da comparação ser entre os dois algoritmos, então é necessário que ambos sejam gerados com base nos mesmos projetos.

Figura 6 – Quantidade de classes que atingiram cobertura de código superior ao outro algoritmo




Fonte: Gustavo da Mota Ramos, 2018

Após a coleta, o conjunto de dados contém a cobertura de código atingida até o tempo máximo registrado para cada classe em cada um dos algoritmos de geração descritos no capítulo Capítulo 3, sendo assim foi possível verificar em qual dos algoritmos cada classe teve maior cobertura e quantificar, como demonstrado na figura 6.

Os resultados de cobertura de código obtidos divergem de alguns resultados apresentados por (SHAMSHIRI *et al.*, ), no experimento que consistiu em gerar os casos de testes a partir de 1000 classes randomicamente escolhidas do corpus SF110. O estudo em questão apontou que seus resultados sugerem que na prática existe pouca diferença quando os resultados de geração de casos de teste são comparados, concluindo que algoritmos aleatórios tenham o mesmo nível de efetividade.

Esta diferença pode ser dada por dois fatores como volume de dados analisado e a evolução da ferramenta, o estudo citado utilizou em torno de de 20% das classes utilizadas para análise na figura 6, o volume maior analisado pode representar uma quantidade maior de casos. Outro fator importante é a evolução da ferramenta EvoSuite ao longo de 3 anos,

por meio de melhorias constantes pode ter causado melhorias no algoritmo genético o suficiente para conseguir o aumento da cobertura de código demonstrado na diferença dos experimentos.

A partir dos dados de cobertura coletados em cada algoritmo, foi possível calcular a diferença entre eles em cada classe. As diferenças foram agrupadas e separadas em diferentes faixas como demonstrado na tabela a seguir:

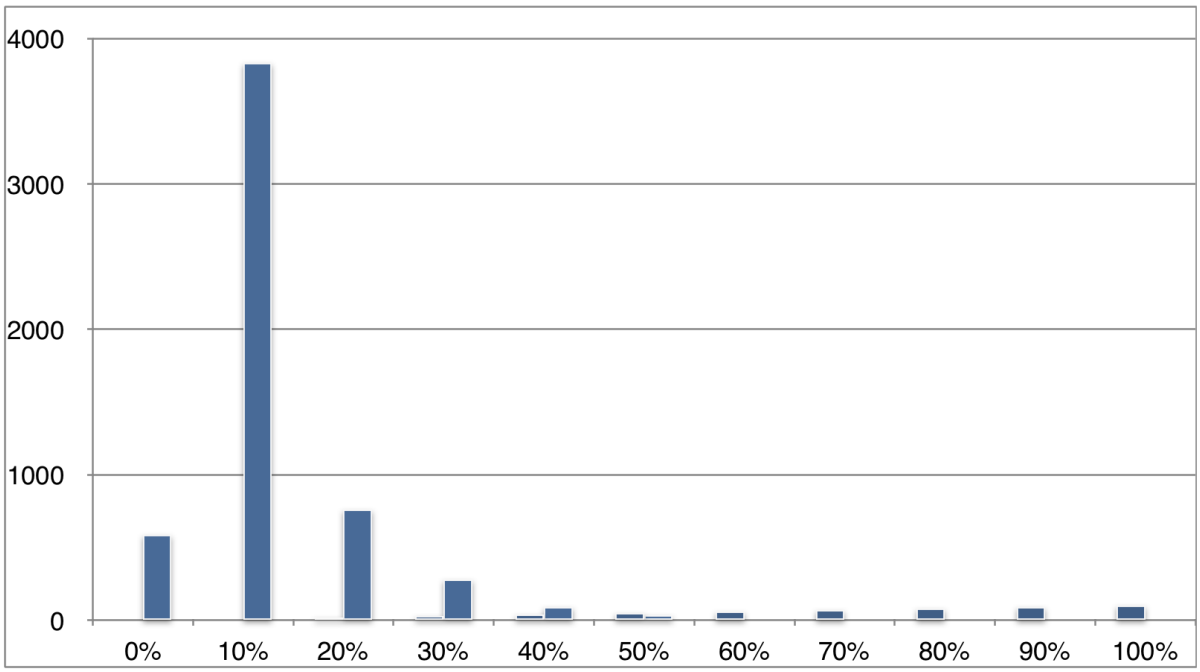
Tabela 2 – Distruibuição das classes nas diferenças de cobertura entre os algoritmos

| DIferença de cobertura | Quantidade de classes |
|---:|---|
| Sem diferença | 585 |
| 10% | 4232 |
| 20% | 1075 |
| 30% | 197 |
| 40% | 74 |
| 50% | 22 |
| 60% | 1 |
| 70% | 3 |
| 80% | 0 |
| 90% | 0 |
| 100% | 2 |

Fonte: Gustavo da Mota Ramos, 2018

O conteúdo da tabela 2 mostra a distruibuição das diferenças de cobertura, é possível notar que maioria das classes, 85.7 % do total, possuem diferença de cobertura entre os algoritmos maior do que 0% e menor do que 20%. A diferença de cobertura entre 0% e 10% pode ser melhor vista na imagem 7.

Figura 7 – Distribuição da cobertura alcançada por cada classe em cada software

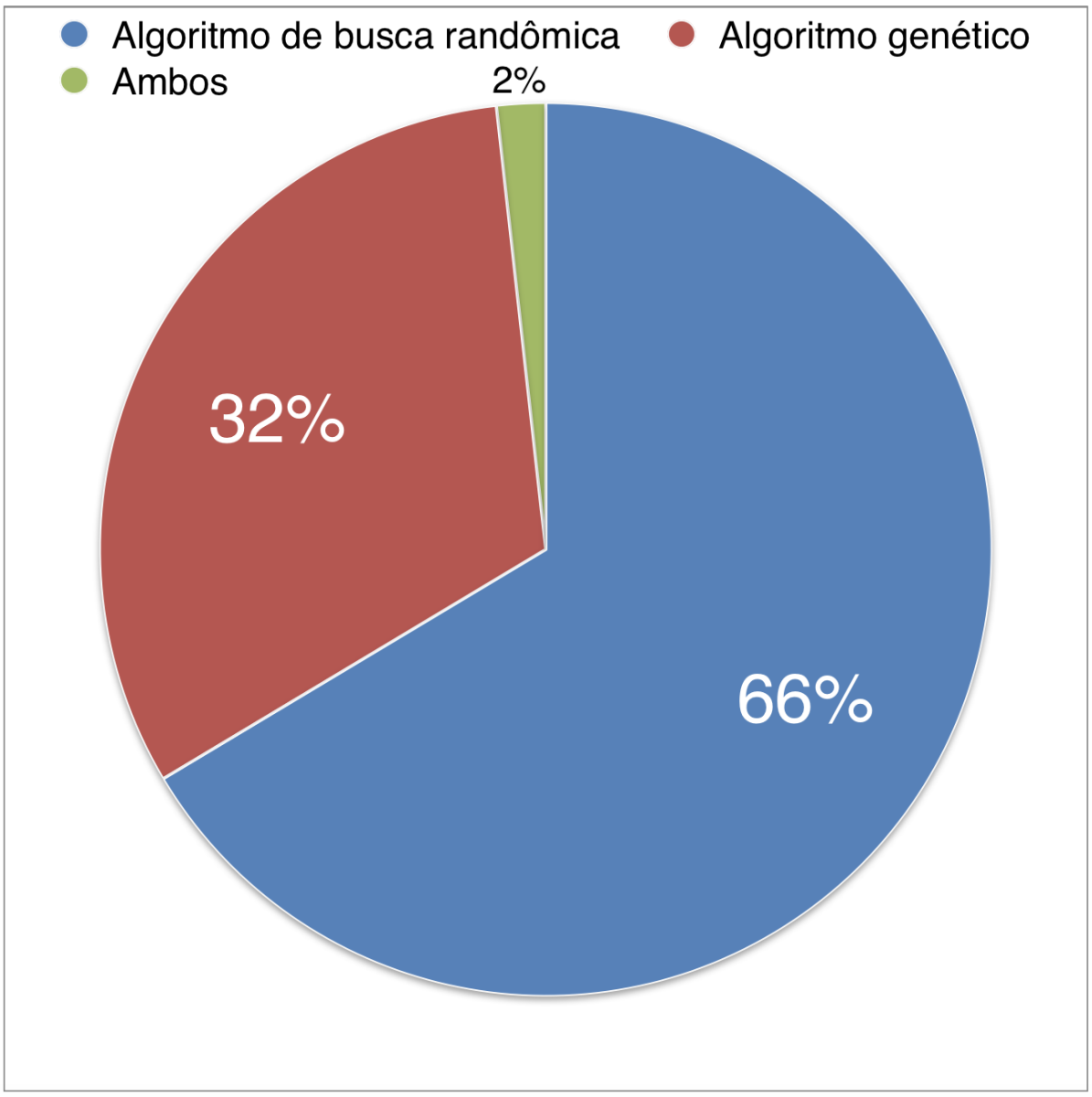



Fonte: Gustavo da Mota Ramos, 2018

A partir da figura 7 é possível notar que a predominância da diferença de cobertura entre as técnicas situa-se entre 0% e 10% . Estes valores são importantes para compreender a magniture da diferença coberta entre os algoritmos para que assim seja possível escolher o algoritmo mais rápido em casos que a diferença seja menor ao construir o conjunto de dados de aprendizado.

Um segundo ponto importante de análise é o tempo de geração utilizado em cada um dos algoritmos, em alguns dos projetos análisados, o tempo de geração total foi superior a 8 horas. o mesmo processo aplicado para análise da cobertura do código foi aplicado para o tempo de geração:

Figura 8 – Quantidade de classes que consumiram menos tempo melhor em cada algoritmo




Fonte: Gustavo da Mota Ramos, 2018

Como pode ser visto na imagem 8, uma quantidade considerável, totalizando 66% das classes foram geradas mais rapidamente utilizando o algoritmo de busca aleatória, enquanto somente 32% das classes tiveram seus dados gerados mais rapidamente utilizando algoritmo genético e somente 2% tiveram seus dados gerados com a mesma quantidade de tempo.

Após a extração das métricas CK os dados foram consolidados por meio de executáveis escritos em Go, os arquivos de entrada consistem nos resultados gerados usando algoritmo genético, resultados gerados usando algoritmo aleatório e as métricas CK. Os 3

arquivos diferem em suas estruturas, porém foram consolidados por meio de umconjunto de dados em comum pelos 3 arquivos: Todos possuem o caminho completo do pacote onde a classe encontra-se e o nome da classe, formando uma tupla única de união das três fontes de dados; O resultado final consistiu em um conjunto de dados de 5705 registros de classes contendo os dados consolidados, resultando assim no cálculo da correlação:

Tabela 3 – Correlação entre métricas CK e cobertura de código

| Métrica | Algoritmo genético | Algoritmo aleatório |
|---|---|---|
| CBO | -0.249850668595503 | -0.232931214471127 |
| WMC | -0.29807261754159 | -0.276826266943643 |
| DIT | -0.221073021178272 | -0.229483879955976 |
| NOC | 0.000302617656848 | 0.001561246261131 |
| RFC | -0.564049275058188 | -0.552823956941685 |
| LCOM | -0.040802466405277 | -0.032051943296475 |
| NOM | -0.109889786736834 | -0.0827808149062 |
| NOPM | -0.014135793315715 | 0.009596758553157 |
| NOSM | -0.075230386707179 | -0.071445190069526 |
| NOF | -0.140224056429869 | -0.120339077987188 |
| NOPF | -0.004633572435886 | -0.000497241219225 |
| NOSF | -0.111343334357608 | -0.109334015568744 |
| NOSI | -0.282084913928777 | -0.273109674212372 |
| LOC | -0.287212833651501 | -0.262779424522538 |

Fonte: Gustavo Ramos, 2018

Tabela 4 – Correlação entre métricas CK e o tempo de geração

| Métrica | Algoritmo genético | Algoritmo aleatório |
|---|---|---|
| CBO | 0.396318591856812 | 0.037898357322244 |
| WMC | 0.484002414944669 | 0.059460693763974 |
| DIT | 0.0040537265288 | 0.005185276134757 |
| NOC | 0.008046221302802 | -0.00048066394787 |
| RFC | 0.491425142687552 | 0.073355181869394 |
| LCOM | 0.186475399697658 | 0.025181662291855 |
| NOM | 0.441803537967141 | 0.038466136152881 |
| NOPM | 0.38853663183873 | 0.027876177461111 |
| NOSM | 0.180912744473895 | 0.030067038091283 |
| NOF | 0.332907657997746 | 0.02963776551629 |
| NOPF | 0.059216533461886 | 0.002387027385852 |
| NOSF | 0.159268739166119 | 0.02786927036709 |
| NOSI | 0.283074420386565 | 0.042066832430523 |
| LOC | 0.476423297500777 | 0.052316363027409 |

Fonte: Gustavo Ramos, 2018

As tabelas 3 e 4 possuem as correlações calculadas entre cada métrica $CK$ extraída com a cobertura atingida nos dois algoritmos. A partir dos resultados presentes nas tabelas é possível notar que os valores de correlação situados entre valores baixos e moderados, enquanto no algoritmo aleatório, com exceção da métrica $RFC$, todos os valores baixos o suficiente para não possuírem qualquer significância para análise.

## *7.1 Aplicação das classes e classificação*

Após a coleta dos dados e consolidação seguido de validação deste conjunto por meio da sua correlação, é possível trabalhar com este conjunto para assim classificar cada um dos elementos. Para a execução de um algoritmo de classificação supervisionado, neste caso o algoritmo naive bayes, é necessário primariamente definir quais são as classes as quais cada elemento do conjunto de dados poderá pertencer, as classes são os nomes dados para os agrupamentos os quais será necessário classificar cada elemento de acordo com suas características, desde que um elemento pertença a uma classe somente.

Dado que o intuito do experimento é reconhecer qual o algoritmo de geração de dados de teste é mais adequado: Algoritmo de busca aleatória ou algoritmo genético, é possível então afirmar que as classes de classificação serão os dois algoritmos, os quais serão tratados com as seguintes classes: **RANDOM** ou **GA** respectivamente. Os conjuntos de treinamento utilizados estão disponíveis na tabela presente no anexo.

### 7.1.1 Conjunto de aprendizado baseado em cobertura de código

O primeiro conjunto de aprendizado montado, levou em consideração somente a cobertura dos testes para montagem do conjunto de aprendizado, ou seja, o tempo foi desconsiderado, sendo assim este conjunto de dados foi nomeado como **CBCK** (Cover based on CK dataset).

**Algoritmo 2** Algoritmo para aplicação das classes no conjunto de treinamento CBCK

```
1: procedure AplicarClasse
2:     Para todas as linhas L do conjunto consolidado:
3:     classe := RANDOM
4:     Se L.CoberturaGA > L.CoberturaRandom: classe := GA
5:     AtualizaArquivo()
```

Fonte: Gustavo Ramos, 2018

O algoritmo 2 demonstra a lógica para aplicação da classe para cada linha do conjunto de testes como é possível ver na linha 2. A partir da linha 3 é possível ver que todas as linhas receberam a classe **RANDOM** , este estado é mudado se a condição da linha 4 seja for satisfeita, ou seja, a linha do conjunto receberá a **GA** se a cobertura com algoritmo genético for maior. A atribuição da linha 3 ocorreu pois além dos casos em que o algoritmo ranômico tenha maior performance, é possível ocorrer o caso em que ambas ambas cobriram a mesma quantidade, estes são os 10% presentes na figura 6. Para os casos em que a cobertura atingida foi igual nos dois algoritmos, a classe **RANDOM** foi aplicada visto que esta consumiu menor tempo na geração de dados como demonstrado na figura 8.

Após a aplicação das classes no conjunto de dados, é possível utilizar o mesmo na ferramenta Weka para assim executar o algoritmo de classificação naive bayes, a própria ferramenta possui um mecanismo onde é possível separar aleatoriamente uma quantidade dos dados do conjunto para treinamento e a parte complementar (usualmente menor) para validação e testes, por padrão foi adotado 80% dos dados para aprendizado enquanto os 20% restantes para testes. O algoritmo foi executado apresentando dois resultados, a precisão e a seguinte matriz de confusão:

Tabela 5 – Matriz de confusão ao classificar o conjunto CBCK com o algoritmo Naive Bayes

| | GA | RANDOM |
|---|---|---|
| GA | 898 | 42 |
| RANDOM | 176 | 25 |

Fonte: Gustavo Ramos, 2018

Os resultados apresentados pela ferramenta demonstram que 78.6% dos casos foram corretamente classificados enquanto 21.4% foram incorretamente classificados. A matriz demonstrada na tabela 5 é chamada de matriz de confusão, as coordenadas na diagonal (Os valores 898 e 25 respectivamente) representam os casos corretamente classificados, ou seja, a linha possuía a classe GA no conjunto de treinamento e foi corretamente classificada como GA, o mesmo casso ocorre para a classe RANDOM. Os outros casos compõe os casos onde uma classe foi dada e o algoritmo classificou em outra classe.

É possível notar, apesar da precisão de 78.6%, que o algoritmo teve maior facilidade de classificar casos onde o algoritmo genético possuiu maior cobertura, porém possui

dificuldade em classificar em torno de 60% dos casos onde o algoritmo aleatório possuiu atingiu maior cobertura.

No conjunto de aprendizado **CBCK**, 14 métricas de softwares orientados a objetos foram utilizadas, como as métricas listadas nas tabelas 3 e 4, porém (XIA; YAN; SI, 2013) afirma que algumas métricas tornam-se redundantes ou irrelevantes em um modelo relacionado à predição de defeitos em softwares.

Os resultados simulados por (XIA; YAN; SI, 2013) provam que diversas métricas são correlacionadas e algumas podem ser úteis em modelos de predição de defeitos, métricas destritas no capitulo 4 como: LOC, RFC, CBO, AMC e CAM. As métricas destacadas coincidem inclusive com as de valores mais altos de correlação na tabela 3.

Dada a relevâncias das métricas citadas, segundo conjunto de aprendizado foi criado com o nome **CBCK+** com base no conjunto de métricas detalhados por (XIA; YAN; SI, 2013), ou seja, somente estas métricas foram selecionadas para execução do algorítmo na interface do software Weka. Dado que as métricas de coesão entre os métodos (CAM) e complexidade média dos métodos não fazem parte do conjunto, estas foram substituídas pelas métricas LCOM e WMC respectivamente, resultando na seguinte matriz de confusão:

Tabela 6 – Matriz de confusão ao classificar o conjunto CBCK com o algoritmo Naive Bayes

| | GA | RANDOM |
|---|---|---|
| GA | 891 | 49 |
| RANDOM | 176 | 25 |

Fonte: Gustavo Ramos, 2018

O algoritmo naive bayes, ao ser executado, retornou uma precisão de 80.3% após a otimização, a partir da matriz de confusão 6 é possível notar que a classificação de algoritmos aleatório possuiu uma melhora modesta de 1.7%.

### 7.1.2 Conjunto de aprendizado híbrido

O terceiro conjunto de aprendizado construído levou em consideração tanto a cobertura dos testes quanto o tempo de execução, a execução do algoritmo foi similar ao algoritmo 2 com uma otimização: Nos casos em que a diferença de cobertura entre os algoritmos situa-se entre 0% e 10%, a classe do algoritmo com o menor tempo de execução foi aplicada, o conjunto de treinamento foi então nomeado por **CBCK10**. O algoritmo

naive bayes foi executado, seguindo o uso das mesmas métricas utilizadas no conjunto de treinamento **CBCK+**, resultando na seguinte matriz de confusão:

Tabela 7 – Matriz de confusão ao classificar o conjunto CBCK10 com o algoritmo Naive Bayes

| | GA | RANDOM |
|---|---|---|
| GA | 524 | 284 |
| RANDOM | 107 | 226 |

Fonte: Gustavo Ramos, 2018

O algoritmo, ao ser executado, retornou uma precisão de 63.5%, porém é possível notar a melhora na distribuição da matriz de confusão demonstrada na tabela 7, o algoritmo passou a reconhecer mais casos onde o algoritmo aleatório deveria ser selecionado enquanto o algoritmo reconheceu uma grande quantidade dos casos onde o algoritmo genético deveria ser selecionado.

### *7.2 Discussão dos resultados*

Neste trabalho foi apresentado um estudo sobre a relação das métricas CK com a cobertura de código e tempo em dois algoritmos de geração de dados de testes distintos. Ao longo do trabalho foram coletados diversos dados sobre ambos os assuntos: métricas e geração de dados de testes, estes dados foram então processados, agrupados e análisados para que assim seja compreendida a relação entre estes, caso exista e possíveis tomadas de decisão.

Nas etapas iniciais de projeto, algumas questões foram levantadas como as questões elicitadas no capítulo 1, estas questões serviram como base para mapear todos os materiais e atividades necessários para garantir respostas. Após as extrações e análises descritas no capítulo 7, é possível assim responder os questionamentos, dentro os questionamentos estão:

> Q1: Existe alguma diferença de performance entre os algoritmos de geração de dados de teste: aleatório e algoritmos genéticos ?

Os algoritmos apresentados possuem diferenças em seus resultados de execução, sejam eles cobertura de código ou tempo de execução. O algoritmo genético apresentou maior porcentagem de cobertura de código em 80% dos casos analisados, estas diferenças situam-se em sua maioria entre 1% e 20% quando comparados às coberturas de código

apresentadas pelo algoritmo aleatório nos testes efetuados. Esta diferença pode ser dada por conta da natureza do algoritmo genético, este é um algoritmo de busca direcionado o qual altera sua própria execução para o problema de busca em análise, enquanto o algoritmo aleatório procura possíveis soluções em um espaço maior de busca, com pouco direcionamento.

Em comparação, o algoritmo aleatório apresentou menor tempo de execução em 66 % dos casos, o tempo de execução reduzido é dado pelo tempo entre as interações do algoritmo, visto que o algoritmo aleatório não possui passos como crossover ou mutação, garantindo assim que consiga varrer o espaço de busca com velocidade maior se comparado ao algoritmo genético.

Sendo assim, é possível concluir que o algoritmo aleatório consegue ser executado com maior velocidade e menor tempo do que um algoritmo genético, porém também é notável que um algoritmo genético consegue atingir maior cobertura de código.

| Q2: Alguma característica ou conjunto de características do software sob teste que faça com que uma técnica usada na geração de casos de teste seja melhor do que outra? |
|---|

Em suma, 14 métricas foram analisadas no contexto de geração de dados de teste para entender características como correlação com o tempo necessário para geração ou a cobertura de código atingida ou efeitos destas métricas no processo de geração. Durante a análise dos resultados foi possível notar que um número maior de métricas pode ser redundante, inclusive danificar o resultado se comparado a um conjunto mais preciso de métricas.

Através a análise foi possível notar também que um conjunto de 5 métricas foram capazes de representar melhor uma classe durante a seleção de algoritmos quando este conjunto foi comparado ao conjunto total de 14 métricas. Neste conjunto de métricas estão: $CBO$ (Coupling between Object Classes), $WMC$ (Weight Method Class ou Complexidade de McCabe), $LCOM$ (Lack of Cohesion of Methods) e $LOC$ (Lines of code) e em especial a métrica $RFC$ (Response for a Class) a qual possuiu maior valor de correlação com a cobertura de código.

O conjunto das 5 métricas citadas apresentou correlação média com a cobertura de código apresentada pelo algoritmo genético, além de demonstrar eficiência no algoritmo de classificação, ao mesmo tempo, as métricas apresentaram pouca correlação com a

cobertura obtida pelo algoritmo aleatório ou com o tempo de execução em ambos os algoritmos, sendo assim, é possível afirmar que um algoritmo genético tem características similares em geração de dados de teste enquanto este fato não ocorre no algoritmo aleatório.

| Q3: É possível escolher entre dois algoritmos de geração de dados de testes a partir de um algoritmo de classificação e métricas de softwares orientados e objetos ? |
|---|

Os resultados apresentados pelos experimentos demonstram que é possível escolher entre os algoritmos de geração utilizados com base em um conjunto reduzido de métricas a uma precisão superior a 70%.

O algoritmo genético conseguiu maior precisão nos testes efetuados, a partir das matrizes de confusão apresentadas é possível notar que a quantidade de instâncias incorretamente classificadas situou-se em torno de 20%, enquanto no caso do algoritmo de busca aleatória, a quantidade de instâncias incorretamente classificadas situou-se em 50%.

É possível que as instâncias corretamente classificadas para uso do algoritmo genético possuam um padrão nas métricas de softwares orientados a objetos enquanto no algoritmo de busca aleatória, o padrão não pode ser descrito por meio de métricas de softwares orientados a objetos ou não exista um padrão claro.

# 8 Conclusão

Os algoritmos de geração de dados de teste baseados em busca, aleatório e baseado em algoritmo genético possuem suas diferenças conceituais e estruturais. Os resultados apresentados pelos experimentos demonstram grandes diferenças tanto na cobertura de código quanto no tempo de execução em cada um dos algoritmos. Algoritmos genéticos, em sua maioria, conseguiram maior cobertura do código a partir dos testes gerados, porém, o algoritmo de busca aleatório conseguiu cobrir uma parcela menor de código com tempo de execução menor.

Dadas as diferenças entre os algoritmos, surge a necessidade de seleção do algoritmo de geração de testes mais adequado para cada classe de um software ou componente de software. Durante o presente trabalho, foi proposto o uso de um algoritmo de classificação, para que a partir de um conjunto completo de dados contendo métricas o classificador pudesse escolher o algoritmo mais adequado para cada caso.

As métricas CK demonstraram relação com a seleção do algoritmo genético, ou seja, o conjunto de métricas CK pode ser utilizado para escolher o algoritmo genético. O algoritmo de busca aleatório também pode ser escolhido a partir de métricas CK, porém, com uma redução razoável da eficácia, sendo assim, recomenda-se o uso de métricas CK para seleção somente de algoritmos genéticos

## *8.1 Trabalhos futuros*

Dada a eficácia da seleção do algoritmo genético a partir de métricas CK, compreende-se então a necessidade de analisar a seleção do algoritmo mais adequado de geração automática de dados de teste com base em métricas CK com um escopo maior. A EvoSuite possui diversos outros algoritmos, com diferentes configurações, como os descritos no capítulo 3.

Como a partir dos resultados foi possível notar que as métricas CK podem ser utilizadas para seleção do algoritmo genético em comparação ao algoritmo aleatório, acredita-se que possa ser possível selecionar entre as variações dos algoritmos genéticos existentes.

## Referências[1]

BASHIR, M. F.; BANURI, S. H. K. Automated model based software test data generation system. In: *Emerging Technologies, 2008. ICET 2008. 4th International Conference on*. [S.l.: s.n.], 2008. p. 275–279. Citado na página 14.

BINDER, R. V. Design for testability in object-oriented systems. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 9, p. 87–101, set. 1994. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/182987.184077⟩. Citado na página 14.

BOUCKAERT, R. R.; FRANK, E.; HALL, M.; KIRKBY, R.; REUTEMANN, P.; SEEWALD, A.; SCUSE, D. Weka manual for version 3-6-0. 12 2008. Citado na página 42.

BRUNTINK, M.; DEURSEN, A. van. Predicting class testability using object-oriented metrics. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. [S.l.: s.n.], 2004. p. 136–145. Citado 3 vezes nas páginas 34, 35 e 36.

CADAR, C.; DUNBAR, D.; ENGLER, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 209–224. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1855741.1855756⟩. Citado na página 14.

CADAR, C.; SEN, K. Symbolic execution for software testing: Three decades later. *Commun. ACM*, ACM, New York, NY, USA, v. 56, n. 2, p. 82–90, fev. 2013. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/2408776.2408795⟩. Citado na página 14.

CAMPOS, J.; GE, Y.; FRASER, G.; ELER, M.; ARCURI, A. An empirical evaluation of evolutionary algorithms for test suite generation. In: MENZIES, T.; PETKE, J. (Ed.). *Search Based Software Engineering*. Cham: Springer International Publishing, 2017. p. 33–48. ISBN 978-3-319-66299-2. Citado 9 vezes nas páginas 15, 24, 25, 26, 27, 28, 29, 30 e 31.

CLASSPATH,JAVA docs. ⟨https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html⟩. Acessado em: 2018-08-04. Nenhuma citação no texto.

DAVE, M.; AGRAWAL, R. Search based techniques and mutation analysis in automatic test case generation: A survey. In: *2015 IEEE International Advance Computing Conference (IACC)*. [S.l.: s.n.], 2015. p. 795–799. Citado na página 24.

DAVIS, M. J. Process and product: Dichotomy or duality? *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 20, n. 2, p. 17–18, abr. 1995. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/224155.565634⟩. Citado na página 18.

DICK, J.; FAIVRE, A. Automating the generation and sequencing of test cases from model-based specifications. In: WOODCOCK, J. C. P.; LARSEN, P. G. (Ed.). *FME '93: Industrial-Strength Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. p. 268–284. Citado na página 14.

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

ELER, M. M.; ENDO, A. T.; DURELLI, V. H. S. An empirical study to quantify the characteristics of java programs that may influence symbolic execution from a unit testing perspective. *Journal of Systems and Software*, v. 121, p. 281–297, 2016. Citado na página 38.

EVOSUITE.ORG. ⟨http://www.evosuite.org/⟩. Acessado em: 2018-07-01. Nenhuma citação no texto.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 416–419. ISBN 978-1-4503-0443-6. Disponível em: ⟨http://doi.acm.org/10.1145/2025113.2025179⟩. Citado 2 vezes nas páginas 14 e 15.

FRASER, G.; ARCURI, A. Whole test suite generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 39, n. 2, p. 276–291, fev. 2013. ISSN 0098-5589. Disponível em: ⟨http://dx.doi.org/10.1109/TSE.2012.14⟩. Citado na página 15.

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 24, n. 2, p. 8:1–8:42, dez. 2014. ISSN 1049-331X. Disponível em: ⟨http://doi.acm.org/10.1145/2685612⟩. Citado na página 15.

FRASER, G.; ROJAS, J. M.; CAMPOS, J.; ARCURI, A. Evosuite at the sbst 2017 tool competition. In: *Proceedings of the 10th International Workshop on Search-Based Software Testing*. Piscataway, NJ, USA: IEEE Press, 2017. (SBST '17), p. 39–41. ISBN 978-1-5386-2789-1. Disponível em: ⟨https://doi.org/10.1109/SBST.2017..6⟩. Citado 3 vezes nas páginas 15, 31 e 32.

FRASER, G.; STAATS, M.; MCMINN, P.; ARCURI, A.; PADBERG, F. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 24, n. 4, p. 23:1–23:49, set. 2015. ISSN 1049-331X. Disponível em: ⟨http://doi.acm.org/10.1145/2699688⟩. Citado na página 15.

GODEFROID, P.; KLARLUND, N.; SEN, K. Dart: Directed automated random testing. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 40, n. 6, p. 213–223, jun. 2005. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/1064978.1065036⟩. Citado na página 14.

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1, n. 2, p. 156–173, June 1975. ISSN 0098-5589. Citado na página 18.

GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008. ISBN 9781844809899. Disponível em: ⟨https://books.google.ie/books?id=Ss62LSqCa1MC⟩. Citado na página 14.

HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 45, n. 1, p. 11:1–11:61, dez. 2012. ISSN 0360-0300. Disponível em: ⟨http://doi.acm.org/10.1145/2379776.2379787⟩. Citado na página 14.

HILBURN, Y.; TOWHIDNEJAD, M. Software quality across the curriculum. In: IEEE. *Frontiers in Education, 2002. FIE 2002. 32nd Annual*. [S.l.], 2002. v. 3, p. S1G–18. Citado na página 14.

JACOB, P. M.; PRASANNA, M. A comparative analysis on black box testing strategies. In: *2016 International Conference on Information Science (ICIS)*. [S.l.: s.n.], 2016. p. 1–6. Citado 2 vezes nas páginas 19 e 21.

JUNIT.ORG/JUNIT5/. ⟨https://junit.org/junit5/⟩. Acessado em: 2018-08-04. Nenhuma citação no texto.

KAN, S. H. *Metrics and Models in Software Quality Engineering*. Reading, MA: Addison Wesley, 1995. Citado 4 vezes nas páginas 33, 34, 36 e 37.

KAPROCKI, Z.; PEKOVIC, V.; VELIKIC, G. Combined testing approach: Increased efficiency of black box testing. In: *2015 IEEE 1st International Workshop on Consumer Electronics (CE WS)*. [S.l.: s.n.], 2015. p. 76–78. Citado na página 20.

KARNOPP, D. C. Random search techniques for optimization problems. *Automatica*, Pergamon Press, Inc., Tarrytown, NY, USA, v. 1, n. 2-3, p. 111–121, ago. 1963. ISSN 0005-1098. Disponível em: ⟨http://dx.doi.org/10.1016/0005-1098(63)90018-9⟩. Citado 2 vezes nas páginas 25 e 26.

KAUR, A.; KAUR, K.; PATHAK, K. Software maintainability prediction by data mining of software code metrics. In: *2014 International Conference on Data Mining and Intelligent Computing (ICDMIC)*. [S.l.: s.n.], 2014. p. 1–6. Citado na página 50.

KOREL, B. Automated software test data generation. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 16, n. 8, p. 870–879, ago. 1990. ISSN 0098-5589. Disponível em: ⟨http://dx.doi.org/10.1109/32.57624⟩. Citado na página 21.

MCMINN, P. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, John Wiley and Sons Ltd., Chichester, UK, v. 14, n. 2, p. 105–156, jun. 2004. ISSN 0960-0833. Disponível em: ⟨http://dx.doi.org/10.1002/stvr.v14:2⟩. Citado na página 14.

NORTHROP, L. M. Let's teach architecting high quality software. In: *19th Conference on Software Engineering Education and Training (CSEE&T 2006), 19-21 April 2006, Turtle Bay, Hawaii, USA*. [s.n.], 2006. p. 5. Disponível em: ⟨http://dx.doi.org/10.1109/CSEET.2006.23⟩. Citado na página 14.

PACHECO, C.; ERNST, M. D. Randoop: Feedback-directed random testing for java. In: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*. [S.l.: s.n.], 2007. p. 815–816. Citado na página 14.

PEZZÈ, M.; YOUNG, M. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008. ISBN 9780471455936. Disponível em: ⟨https://books.google.com.br/books?id=mjEiAQAAIAAJ⟩. Citado na página 14.

PFLEEGER, S.; ATLEE, J. *Software Engineering: Theory and Practice*. Prentice Hall, 2010. ISBN 9780136061694. Disponível em: ⟨https://books.google.com.br/books?id=7zbSZ54JG1wC⟩. Citado na página 18.

PRESSMAN, R. *Engenharia de Software - 7.ed.:*. McGraw Hill Brasil, 2009. ISBN 9788580550443. Disponível em: ⟨https://books.google.com.br/books?id=y0rH9wuXe68C⟩. Citado 5 vezes nas páginas 18, 21, 22, 23 e 37.

REPASI, T. Software testing - state of the art and current research challanges. In: *2009 5th International Symposium on Applied Computational Intelligence and Informatics*. [S.l.: s.n.], 2009. p. 47–50. Citado na página 21.

ROJAS, J. M.; FRASER, G. Is search-based unit test generation research stuck in a local optimum? In: *2017 IEEE ACM 10th International Workshop on Search-Based Software Testing (SBST)*. [S.l.: s.n.], 2017. p. 51–52. Citado na página 25.

ROJAS, J. M.; VIVANTI, M.; ARCURI, A.; FRASER, G. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, v. 22, n. 2, p. 852–893, Apr 2017. Disponível em: ⟨https://doi.org/10.1007/s10664-015-9424-2⟩. Citado na página 15.

SEN, K.; AGHA, G. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In: BALL, T.; JONES, R. B. (Ed.). *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 419–423. ISBN 978-3-540-37411-4. Citado na página 14.

SEN, K.; MARINOV, D.; AGHA, G. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 5, p. 263–272, set. 2005. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1095430.1081750⟩. Citado na página 14.

SETH, H. R.; BANKA, H. Hardware implementation of naïve bayes classifier: A cost effective technique. In: *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*. [S.l.: s.n.], 2016. p. 264–267. Citado 3 vezes nas páginas 39, 40 e 41.

SHAMSHIRI, S.; ROJAS, J. M.; GALEOTTI, J. P.; WALKINSHAW, N.; FRASER, G. How do automatically generated unit tests influence software maintenance? In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2018. p. 250–261. Citado na página 24.

SHAMSHIRI, S.; ROJAS, J. M.; GAZZOLA, L.; FRASER, G.; MCMINN, P.; MARIANI, L.; ARCURI, A. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability*, v. 28, n. 4, p. e1660. E1660 stvr.1660. Disponível em: ⟨https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1660⟩. Citado 7 vezes nas páginas 24, 26, 27, 43, 46, 47 e 53.

SHARMA, P.; SINGH, D.; SINGH, A. Classification algorithms on a large continuous random dataset using rapid miner tool. In: *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*. [S.l.: s.n.], 2015. p. 704–709. Citado 3 vezes nas páginas 39, 40 e 41.

SINGH, R. Test case generation for object-oriented systems: A review. In: *2014 Fourth International Conference on Communication Systems and Network Technologies*. [S.l.: s.n.], 2014. p. 981–989. Citado na página 19.

SITE.MOCKITO.ORG. ⟨http://site.mockito.org⟩. Acessado em: 2018-08-04. Nenhuma citação no texto.

SLAUGHTER, S. A.; HARTER, D. E.; KRISHNAN, M. S. Evaluating the cost of software quality. *Commun. ACM*, ACM, New York, NY, USA, v. 41, n. 8, p. 67–73, ago. 1998. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/280324.280335⟩. Citado 2 vezes nas páginas 21 e 22.

SOMMERVILLE, I. *Software Engineering*. 9th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0137035152, 9780137035151. Citado 2 vezes nas páginas 18 e 19.

SOMMERVILLE, I.; MELNIKOFF, S.; ARAKAKI, R.; BARBOSA, E. de A. *Engenharia de software*. ADDISON WESLEY BRA, 2008. ISBN 9788588639287. Disponível em: ⟨https://books.google.com.br/books?id=ifIYOgAACAAJ⟩. Citado na página 14.

SPADINI, D.; ANICHE, M.; BRUNTINK, M.; BACCHELLI, A. To mock or not to mock? an empirical study on mocking practices. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. p. 402–412. Citado 2 vezes nas páginas 19 e 20.

SRIVASTAVA, S.; KUMAR, R. Indirect method to measure software quality using ck-oo suite. In: *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*. [S.l.: s.n.], 2013. p. 47–51. Citado 3 vezes nas páginas 33, 34 e 37.

SUBBULAKSHMI, C. V.; DEEPA, S. N.; MALATHI, N. Comparative analysis of xlminer and weka for pattern classification. In: *2012 IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. [S.l.: s.n.], 2012. p. 453–457. Citado na página 42.

SURESH, Y. Software quality assessment for open source software using logistic amp; naive bayes classifier. In: *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. [S.l.: s.n.], 2016. p. 267–272. Citado 2 vezes nas páginas 37 e 40.

TAHIR, A.; MACDONELL, S. G.; BUCHAN, J. Understanding class-level testability through dynamic analysis. In: *Evaluation of Novel Approaches to Software Engineering (ENASE), 2014 International Conference on*. [S.l.: s.n.], 2014. p. 1–10. Citado na página 14.

TILLMANN, N.; HALLEUX, J. de. Pex–white box test generation for .net. In: BECKERT, B.; HÄHNLE, R. (Ed.). *Tests and Proofs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 134–153. ISBN 978-3-540-79124-9. Citado na página 14.

VISSER, W.; PăSăREANU, C. S.; KHURSHID, S. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 4, p. 97–107, jul. 2004. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1013886.1007526⟩. Citado na página 14.

WAGNER, S.; SEIFERT, T. Software quality economics for defect-detection techniques using failure prediction. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 30, n. 4, p. 1–6, maio 2005. ISSN 0163-5948. Disponível em: ⟨http://doi.acm.org/10.1145/1082983.1083296⟩. Citado 3 vezes nas páginas 21, 22 e 23.

WALKINSHAW, N.; FRASER, G. Uncertainty-driven black-box test data generation. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. [S.l.: s.n.], 2017. p. 253–263. Citado na página 21.

WATSON, A. H.; MCCABE, T. J.; WALLACE, D. R. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In: *U.S. Department of Commerce/N ational Institute of Standards and Technology*. [S.l.: s.n.], 1996. Citado na página 34.

XIA, Y.; YAN, G.; SI, Q. A study on the significance of software metrics in defect prediction. In: *2013 Sixth International Symposium on Computational Intelligence and Design*. [S.l.: s.n.], 2013. v. 2, p. 343–346. Citado na página 60.

XU, S.; CHEN, L.; WANG, C.; RUD, O. A comparative study on black-box testing with open source applications. In: *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. [S.l.: s.n.], 2016. p. 527–532. Citado na página 21.

ZEISS, B.; VEGA, D.; SCHIEFERDECKER, I.; NEUKIRCHEN, H.; GRABOWSKI, J. Applying the iso 9126 quality model to test specifications - exemplified for ttcn-3 test specifications. In: *Software Engineering*. [S.l.: s.n.], 2007. Citado na página 33.

# Anexo A – Conjuntos de treinamento

Tabela 8 – Conjuntos de treinamento utilizados

| Conjunto de treinamento | CSV | Formato Weka |
|---|---|---|
| CBCK | Download | Download |
| CBCK+ | - | Download |
| CBCK10 | Download | Download |

Fonte: Gustavo Ramos, 2018