



UNIVERSIDADE DE SÃO PAULO

Escola de Artes, Ciências e Humanidades

EVERTON NOTE NARCISO

**SELEÇÃO DE CASOS DE TESTE PARA  
SISTÉMAS DE PROCESSAMENTO DE  
IMAGENS UTILIZANDO CONCEITOS  
DE CBIR**

SÃO PAULO  
Agosto de 2013

EVERTON NOTE NARCISO

# **SELEÇÃO DE CASOS DE TESTE PARA SISTEMAS DE PROCESSAMENTO DE IMAGENS UTILIZANDO CONCEITOS DE CBIR**

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

Versão corrigida contendo as alterações solicitadas pela comissão julgadora. A versão original encontra-se disponível na Escola de Artes, Ciências e Humanidades.

Orientador(a): Prof<sup>a</sup>. Dr<sup>a</sup>. Fátima de Lourdes dos Santos Nunes Marques.

SÃO PAULO  
Agosto de 2013

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

CATALOGAÇÃO-NA-PUBLICAÇÃO

Biblioteca

Escola de Artes, Ciências e Humanidades da Universidade de São Paulo

Narciso, Everton Note

Seleção de casos de teste para sistemas de processamento de imagens utilizando conceitos de CBIR / Everton Note Narciso; orientador, Fátima de Lourdes dos Santos Nunes Marques – São Paulo, 2013.

127 f.

Dissertação (Mestrado em Ciências) – Programa de Pós-Graduação em Sistemas de Informação, Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, em 2013.

Versão corrigida.

1. Teste e avaliação de software.
2. Sistemas de informação.
3. Processamento de imagens.
4. Engenharia de software. I. Marques, Fátima de Lourdes dos Santos Nunes, orient. II. Título.

CDD 22.ed. – 005.14

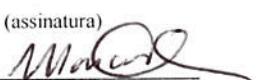
## ATA DE DEFESA DE DISSERTAÇÃO

A Comissão Julgadora da defesa de dissertação de mestrado reuniu-se na data de 29 de outubro de 2013, às 14:00, na Escola de Artes, Ciências e Humanidades da Universidade de São Paulo, São Paulo (SP), para arguir, em sessão pública, o(a) aluno(a) **Everton Note Narciso**, do Programa de Pós-Graduação em Sistemas de Informação (Mestrado), referente à dissertação intitulada “Seleção de Casos de Teste para Sistemas de Processamento de Imagens Utilizando Conceitos de CBIR”. A sessão iniciou-se com a apresentação do projeto pelo aluno, seguida de arguição pelos examinadores, que decidiram pela APROVAÇÃO (aprovação/reprovação) do aluno na defesa de dissertação de mestrado, conforme avaliação dos examinadores a seguir.

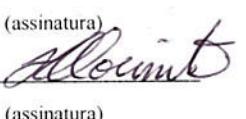
Profº. Drª. Fátima de Lourdes dos S. N. Marques Resultado: APROVADO

  
(assinatura)

Prof. Dr. Marcos Lordello Chaim Resultado: APROVADO

  
(assinatura)

Prof. Dr. Carlos Hitoshi Morimoto Resultado: Aprovado

  
(assinatura)

Aprovado pela CCP/PGSI  
em 08/11/2013

São Paulo, 29 de outubro de 2013.

Aprovado pela CPG/EACH  
em 11/11/2013

  
Profº Drª Fátima de Lourdes dos Santos Nunes Marques  
Coordenadora de Programa  
Sistemas de Informação  
EACH-USP

Dedico este trabalho a toda a minha família, aos professores do Programa de Pós-Graduação em Sistemas de Informação da EACH/USP e a todos os meus amigos.

Agradeço a minha família, a minha noiva Alyne Corrêa de Freitas, a minha orientadora Profª. Drª. Fátima de Lourdes dos Santos Nunes Marques, aos professores Adilson Ferreira da Silva e João Carlos Lopes Fernandes (IESA/Fatec), José Osvaldo Couto Horta (Universidade Nove de Julho), Delhi Tereza Paiva Salinas e Marcelo de Souza Lauretto (EACH/USP) e ao colega Vagner M. Gonçalves (EACH/USP) pela ajuda e incentivo durante o decorrer do curso.

Acredito que somente uma pessoa que nada aprendeu  
não modifica suas opiniões.  
(Emil Zatopek)

# ***Resumo***

NARCISO, Everton Note. **Seleção de Casos de Teste para Sistemas de Processamento de Imagens Utilizando Conceitos de CBIR.** 2013. 127 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2013.

Os sistemas de processamento de imagens exercem um papel importante no que tange à emulação da visão humana, pois grande parte das informações que as pessoas obtém do mundo real ocorre por meio de imagens. Desenvolver tais sistemas é uma tarefa complexa e que requer testes rigorosos para garantir a sua confiabilidade. Neste cenário, a seleção de casos de teste é fundamental, pois ajuda a eliminar os dados de teste redundantes e desnecessários enquanto procura manter altas taxas de detecção de erros. Na literatura há várias abordagens para seleção de casos de teste com foco em sistemas de entradas/saídas alfanuméricas, mas a seleção voltada a sistemas complexos (e.g. processamento de imagens) ainda é pouco explorada. Visando a contribuir neste campo de pesquisa, este trabalho apresenta um novo método intitulado *Tcs&CbIR*, que seleciona e recupera um subconjunto de imagens a partir de um vasto conjunto de teste. Os testes realizados com dois programas de processamento de imagens mostram que a nova abordagem pode superar a seleção aleatória pois, no contexto de avaliação apresentado, a quantidade de casos de teste necessária para revelar a presença de erros foi reduzida em até 87%. Os resultados obtidos revelam, também, o potencial da utilização de CBIR para abstração de informações, a importância da definição de extratores de características adequados e a influência que as funções de similaridade podem exercer na seleção de casos de teste.

**Palavras-chave:** Seleção de Casos de Teste, Sistemas de Processamento de Imagens, CBIR.

# ***Abstract***

NARCISO, Everton Note. **Test Case Selection For Image Processing Systems Using CBIR Concepts.** 2013. 127 p. Dissertation (Master in Sciences) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2013.

Image processing systems play a key role when it comes to emulation of human vision, because much of the information that humans capture from the real world occurs through images. Developing such systems is a complex task that requires rigorous testing to ensure their quality and reliability. In this scenario, the test case selection is crucial because it helps to eliminate the redundant and unnecessary test data while it tries to maintain high rates of error detection. In the literature there are several approaches for test cases selection with a focus on systems with alphanumeric inputs and outputs, but the selection focused on complex systems (e.g. image processing) is still unexplored. Aiming to contribute to this research field, this work presents a new method entitled *Tcs&CbIR*, which selects and retrieves a subset of images from a wide test suite. Tests conducted with two image processing programs show that the new approach can overcome the random selection because, in the context of evaluation presented, the amount of test cases required to detect the presence of the errors was reduced by up to 87%. The results also show the potential use of CBIR for information abstraction, the importance of the definition of suitable extractors of characteristics and the influence of the similarity functions in the test case selection.

**Keywords:** Test Case Selection, Image Processing Systems, CBIR.

# ***Lista de Figuras***

1	Representação de uma imagem por meio da matriz de <i>pixels</i> . . . . .	5
2	Representação da interação entre diversas áreas de sistemas de imagens digitais . . . . .	6
3	Representação do modelo RGB . . . . .	7
4	Representação de imagens no padrão RGB por meio de vetores . . . . .	7
5	Estrutura típica de um sistema de CBIR . . . . .	8
6	Casos de teste: os diferentes parâmetros de entrada e os passos para se testar um <i>software</i> . . . . .	11
7	Representação simplificada de um grafo de fluxo de controle . . . . .	14
8	Arquitetura do <i>framework</i> O-FIm . . . . .	17
9	Interface principal do <i>framework</i> O-FIm . . . . .	18
10	Particionamento do conjunto de testes . . . . .	19
11	Representação diagramática do processo de <i>clustering</i> . . . . .	21
12	Representação do processo de AHC . . . . .	22
13	Dendograma do processo de <i>clustering</i> , método AHC . . . . .	25
14	Processo da pesquisa em bases de dados científicas . . . . .	31
15	As heurísticas de seleção de casos de teste mais abordadas . . . . .	34
16	Métodos mais utilizados em seleção de casos de teste . . . . .	36
17	Relação entre as heurísticas e os métodos de seleção de casos de teste . . .	42
18	Seleção de casos de teste utilizando conceitos de CBIR . . . . .	52
19	Exemplo real de extração das características área, perímetro e cor . . . .	55
20	Resultado de saída da etapa de extração de características . . . . .	55

21	Indexação e armazenamento dos vetores de características . . . . .	56
22	Otimização da matriz de similaridade . . . . .	57
23	Implementação do <i>clustering AHC</i> . . . . .	57
24	Etapa de seleção e recuperação de imagens . . . . .	58
25	Estrutura de implementação do <i>Tcs&amp;CbIR</i> . . . . .	59
26	Ilustração do processo de detecção de borda utilizando funções de gradiente	62
27	Ilustração do processo de esqueletização . . . . .	62
28	Imagens capazes de revelar a presença de erros nos SUT1, SUT2 e SUT3 .	65
29	Processo de execução dos testes . . . . .	68
30	<i>Scatterplot</i> dos extractores cor, área e perímetro . . . . .	70
31	Resultado do processo de <i>clustering</i> . . . . .	74
32	Histogramas dos testes realizados com o SUT1 . . . . .	75
33	Histogramas dos testes realizados com o SUT2 . . . . .	76
34	Histogramas dos testes realizados com o SUT3 . . . . .	76
35	<i>Boxplots</i> e histogramas sobrepostos dos testes realizados com o SUT1 . .	77
36	<i>Boxplots</i> e histogramas sobrepostos dos testes realizados com o SUT2 . .	78
37	<i>Boxplots</i> e histogramas sobrepostos dos testes realizados com o SUT3 . .	78
38	Exemplo de um histograma utilizado em análises estatísticas . . . . .	108
39	Ilustração do gráfico <i>boxplot</i> . . . . .	109
40	Ilustração do gráfico <i>scatterplot</i> . . . . .	110

# ***Lista de Tabelas***

1	Representação dos níveis de processamento de imagens digitais . . . . .	5
2	Renda e idade de seis indivíduos distintos . . . . .	22
3	Resultados obtidos após a execução do algoritmo AHC . . . . .	25
4	Quantidade de artigos obtidos nas bases de dados indexadas . . . . .	31
5	Principais veículos de publicação . . . . .	32
6	Pesquisadores e suas publicações . . . . .	32
7	Visão global dos métodos de seleção de casos de teste . . . . .	36
8	Tipos de programas utilizados em avaliações dos métodos de seleção . . . .	43
9	Principais métricas para avaliação de métodos de seleção de casos de teste	43
10	Principais funcionalidades dos pacotes que compõem o <i>Tcs&amp;CbIR</i> . . . .	60
11	Erros semeados nos SUT1, SUT2 e SUT3 . . . . .	64
12	Testes de correlação . . . . .	71
13	Resultados obtidos utilizando as medidas F-measure, $F_{Tcs\&CbIR}$ e $F_{rate}$ . .	75
14	Resultados dos histogramas e dos cálculos de variabilidade . . . . .	77

# ***Lista de Algoritmos***

1	<i>K-Means</i>	27
2	<i>Adaptive random testing</i>	37
3	Genético	39
4	<i>Greedy</i>	40
5	Extrator de característica “área” para imagens em tons de cinza	53
6	Extrator de característica “cor” para imagens em padrão RGB	54
7	Extrator de característica “perímetro” para imagens binárias	54
8	Procedimento para encontrar os casos de teste capazes de revelar a presença de erros	65
9	Código-fonte do <i>SoftBorda</i> .	94
10	Código-fonte do <i>SoftSkeleton</i>	98
11	Código-fonte das funções de similaridade.	99
12	Código-fonte das funções estatísticas.	101
13	Código-fonte das funções de processamento de imagens.	103
14	Código-fonte das funções de <i>clustering</i> .	106

# *Sumário*

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Motivação . . . . .	2
1.2	Objetivos e contribuições . . . . .	2
1.3	Organização do trabalho . . . . .	3
<b>2</b>	<b>Conceitos Fundamentais</b>	<b>4</b>
2.1	Processamento de imagens digitais . . . . .	4
2.2	O modelo de cores RGB . . . . .	6
2.3	Recuperação de imagens com base em conteúdo (CBIR) . . . . .	8
2.3.1	Estágios do CBIR . . . . .	9
2.3.2	Extratores de características e funções de similaridade . . . . .	9
2.3.3	Estruturas de indexação . . . . .	10
2.4	Teste de <i>software</i> . . . . .	10
2.4.1	Casos de teste . . . . .	11
2.4.2	Teste funcional . . . . .	12
2.4.3	Teste baseado em defeitos . . . . .	12
2.4.3.1	Semeadura de erros ( <i>error seeding</i> ) . . . . .	13
2.4.3.2	Testes de mutação ( <i>mutation testing</i> ) . . . . .	13
2.4.4	Teste estrutural . . . . .	14
2.4.5	Oráculos de teste . . . . .	15
2.5	O <i>framework O-FIm</i> ( <i>Oracle for Images</i> ) . . . . .	16
2.6	Seleção de casos de teste . . . . .	18

2.7	<i>Clustering</i>	21
2.7.1	Método aglomerativo hierárquico (AHC)	22
2.7.2	Método <i>K-Means</i>	27
2.8	Considerações finais	28
<b>3</b>	<b>Revisão Bibliográfica Sistemática</b>	<b>29</b>
3.1	Seleção de trabalhos	29
3.2	Condução da revisão sistemática	30
3.3	Resultados e discussões	31
3.3.1	Visão geral das heurísticas de seleção de casos de teste	33
3.3.1.1	Heurística de seleção aleatória	34
3.3.1.2	Heurística de seleção utilizando conceitos de similaridade	34
3.3.1.3	Heurística de seleção utilizando critérios de cobertura	35
3.3.2	Visão geral dos métodos de seleção de casos de teste	35
3.3.2.1	Método <i>adaptive random testing</i> (ART)	36
3.3.2.2	Método <i>similarity-based selection</i>	38
3.3.2.3	Métodos <i>search-based</i>	39
3.3.2.4	Métodos de <i>clustering</i>	41
3.3.2.5	Métodos estatísticos e lógica <i>Fuzzy</i>	41
3.3.3	Relação entre métodos e heurísticas de seleção de casos de teste	42
3.3.4	Avaliação dos métodos de seleção de casos de teste	42
3.4	Trabalhos relacionados	44
3.5	Tendências e desafios	45
3.6	Limitações da pesquisa	46
3.7	Conclusões da revisão sistemática	47
3.8	Considerações finais	47

<b>4 Seleção de Casos de Teste Utilizando Conceitos de CBIR</b>	<b>49</b>
4.1 Descrição do problema . . . . .	49
4.2 Metodologia e definição da proposta - <i>Tcs&amp;CbIR</i> . . . . .	52
4.2.1 Extratores de características . . . . .	53
4.2.2 Indexação e armazenamento . . . . .	55
4.2.3 Funções de similaridade . . . . .	56
4.2.4 Seleção e recuperação de imagens . . . . .	57
4.2.5 Visão geral de implementação . . . . .	58
4.3 Considerações finais . . . . .	60
<b>5 Plano de Teste</b>	<b>61</b>
5.1 Tecnologias utilizadas . . . . .	61
5.2 <i>Software</i> de processamento de imagens . . . . .	61
5.3 Pré-processamento do conjunto de testes . . . . .	63
5.4 <i>Software</i> sob teste . . . . .	64
5.5 Métricas de avaliação . . . . .	66
5.6 Metodologia de avaliação . . . . .	67
5.7 Execução dos testes e saídas esperadas . . . . .	68
5.8 Implementação e pré-avaliação dos extratores de características . . . . .	69
5.9 Considerações finais . . . . .	71
<b>6 Resultados e Discussões</b>	<b>72</b>
6.1 Descrição geral dos testes e questões de pesquisa . . . . .	72
6.2 Análise de <i>clustering</i> . . . . .	73
6.3 Resultados obtidos . . . . .	74
6.4 Discussões . . . . .	80
6.5 Considerações finais . . . . .	81

<b>7 Conclusões</b>	<b>82</b>
7.1 Trabalhos futuros . . . . .	83
7.2 Publicações . . . . .	84
<b>Referências</b>	<b>85</b>
<b>Apêndice A Código-Fonte do <i>SoftBorda</i></b>	<b>92</b>
<b>Apêndice B Código-Fonte do <i>SoftSkeleton</i></b>	<b>95</b>
<b>Apêndice C Código-Fonte dos Pacotes, Classes e Funções Implementados</b>	<b>99</b>
<b>Apêndice D Métodos Estatísticos</b>	<b>107</b>

# ***Capítulo 1***

## ***Introdução***

Os sistemas de processamento de imagens são tecnologias computacionais cada vez mais presentes nas rotinas organizacionais e pessoais da sociedade moderna, pois grande parte das informações que o ser humano obtém do mundo que o cerca ocorre por meio de imagens. Esses sistemas são compostos por um conjunto de técnicas para capturar, representar e transformar imagens com o auxílio do computador [1]. Um exemplo notável capaz de descrever a importância desses sistemas é a sua aplicação no auxílio a diagnósticos médicos assistidos por computador [2].

Este tipo de sistema normalmente requer elementos capazes de processar e recuperar as imagens de modo eficiente, bem como uma grande capacidade de armazenamento. Logo, desenvolver um *software* de processamento de imagens confiável e de qualidade não é uma tarefa trivial. A complexidade inerente à natureza dos problemas abordados e a necessidade de atender aos requisitos de *software* torna o seu desenvolvimento difícil e requer a aplicação de técnicas provenientes das mais diversas áreas do conhecimento.

Delamaro et al. [3] ressaltam que o processo de desenvolvimento de *software* envolve uma série de atividades que, apesar das técnicas, métodos e ferramentas empregados, não consegue garantir a construção de um produto sem erros. Por isso, atividades agregadas sob o nome de Garantia de Qualidade de *Software* têm sido introduzidas ao longo de todo o processo.

O teste é uma questão fundamental em termos de qualidade do *software*. As metodologias utilizadas durante a fase de testes têm impacto direto na confiabilidade do produto. No entanto, suas especificações frequentemente são negligenciadas e os obstáculos impostos a sua prática são muitos, como o planejamento e a execução inadequados, redução de tempo e custos indiscriminados, a condução de atividades de forma manual e subjetiva, dentre outros [4].

Segundo Sommerville [5], as ferramentas de automatização de testes estão entre as mais importantes a serem desenvolvidas na área de Engenharia de Software. A automatização dos testes envolve processos que visam a minimizar a subjetividade dos testes manuais e a otimizar os recursos disponíveis. Em qualquer organização, os recursos são

cruciais e necessitam ser gerenciados; portanto, criar mecanismos que explorem os requisitos do *software* com o menor esforço computacional possível é um grande desafio.

Nesse cenário, desenvolver métodos e ferramentas para seleção de casos de teste (SCT) é muito importante para a definição das estratégias de teste [4]. A seleção de casos de teste consiste em técnicas para selecionar dados de teste considerando as condições de execução e os resultados esperados para testar o *software* [6]. Os casos de teste podem ser utilizados para atingir diferentes objetivos, como verificar a conformidade com os requisitos, avaliar a robustez em condições de *stress*, o desempenho ou a usabilidade do *software* [7]. Em consonância com esses objetivos, a seleção de casos de teste visa a suprimir os dados redundantes ou desnecessários e a proporcionar altas taxas de detecção de falhas, contribuindo para o aprimoramento do processo de testes.

Em linhas gerais, testar um *software* de processamento de imagens utilizando uma enorme quantidade de imagens (casos de teste) exige um alto custo computacional. Considerando a inviabilidade dessa abordagem, o presente trabalho apresenta um novo método intitulado *Tcs&CbIR*, que utiliza conceitos de CBIR (*Content-Based Image Retrieval*) para selecionar um subconjunto otimizado de imagens dissímiplares a partir de um vasto conjunto de testes.

## 1.1 Motivação

A revisão bibliográfica sistemática (RS) apresentada no capítulo 3 mostra que existem muitos métodos de seleção de casos de teste para *software* embarcado e *software* sob o paradigma de orientação a objetos em geral, normalmente com entradas/saídas alfanuméricas. No entanto, há uma grande lacuna quando se trata de aplicações para *software* de domínio gráfico, como os sistemas de processamento de imagens. Esta lacuna é a principal motivação deste trabalho, pois essa área do conhecimento, apesar de pouco explorada, certamente contribui para a melhoria da qualidade do *software* por meio do aprimoramento das atividades de teste.

## 1.2 Objetivos e contribuições

O presente trabalho tem como objetivo a definição e a implementação de um método que possibilite a seleção automatizada de casos de teste (imagens) para sistemas de processamento de imagens, visando a diminuir o tempo, o custo e a subjetividade dos testes.

Essa abordagem é inovadora e envolve diferentes aspectos técnicos e de pesquisa. As principais contribuições para a área de Sistemas de Informação são: o desenvolvimento de uma nova abordagem para a seleção de casos de teste para programas de processamento de imagens utilizando conceitos de CBIR, bem como o estabelecimento de metodologias para análise e validação dos resultados da abordagem proposta.

Os resultados obtidos mostram o potencial da utilização de CBIR para a seleção de casos de teste. Três estudos experimentais realizados em dois programas de processamento de imagens distintos mostram que a abordagem proposta, quando comparada com a seleção aleatória, pode melhorar significativamente o desempenho de testes por meio da redução do custo computacional necessário para revelar a presença dos erros<sup>†</sup> contidos no *software* sob teste (SUT, do inglês *Software Under Test*).

### 1.3 Organização do trabalho

Para atingir o objetivo proposto, o trabalho foi organizado, após o capítulo introdutório, da seguinte maneira:

- No capítulo 2 são apresentados os conceitos fundamentais, tais como o processamento de imagens digitais, o modelo de cores RGB, recuperação de imagens baseada em conteúdo (CBIR), teste de *software* e *clustering*;
- O capítulo 3 mostra a revisão bibliográfica e aborda os principais métodos de seleção de casos de teste encontrados na literatura, bem como as métricas de avaliação dos métodos reportados;
- No capítulo 4 se discute a metodologia, a definição e a implementação da abordagem proposta (*Tcs&CbIR*);
- O capítulo 5 descreve o plano de teste, projetado para demonstrar a aplicabilidade e avaliar a eficácia do *Tcs&CbIR*;
- No capítulo 6 são apresentados os resultados, as análises e as discussões necessárias para a validação da proposta e
- O capítulo 7 conclui o presente trabalho.

---

<sup>†</sup>No presente trabalho não se adotou qualquer distinção entre os termos defeito, erro ou falha, pois os problemas que acarretam o mal funcionamento do *software* ou o momento em que os problemas foram encontrados não tem influência no termo utilizado para descrevê-lo.

# *Capítulo 2*

## *Conceitos Fundamentais*

Para o desenvolvimento deste trabalho é indispensável somar as contribuições provenientes de diversas áreas da Computação, tais como o processamento de imagens, *Content-based Image Retrieval* e Engenharia de Software. Com o objetivo de apresentar os principais conceitos necessários para a execução da proposta, a composição deste capítulo foi realizada da seguinte maneira: a seção 2.1 apresenta os principais conceitos de processamento de imagens; a seção 2.2 descreve o modelo de cores RGB; a seção 2.3 apresenta os conceitos de recuperação de imagens baseada em conteúdo (CBIR); a seção 2.4 apresenta conceitos fundamentais de teste de *software*; a seção 2.5 descreve o *framework O-FIm*; na seção 2.6 se discute a problemática de seleção de casos de teste; a seção 2.7 apresenta técnicas de *clustering* e a seção 2.8 conclui o capítulo.

### **2.1 Processamento de imagens digitais**

De acordo com Conci et al. [8], até 1980 o maior emprego das imagens digitais consistia em imagens provenientes da pesquisa espacial. Atualmente, essas imagens digitais se encontram difundidas em muitas aplicações. O aumento de seu emprego se deve ao fato de que grande parte das informações que o ser humano obtém do mundo que o cerca ocorre por meio de imagens, tornando necessária a pesquisa e o desenvolvimento em diversas áreas do conhecimento.

O processamento digital de imagens consiste em um conjunto de técnicas para capturar, representar e transformar imagens com o auxílio do computador [1]. A imagem digital é uma representação de uma cena em uma região discreta, limitada por um conjunto finito de valores inteiros que representam cada um dos seus pontos [8]. Matematicamente, uma imagem pode ser descrita como uma função de intensidade de luz refletida por um objeto [2]. No domínio espacial, uma imagem bidimensional pode ser definida como uma função  $f(x,y)$ , onde  $x$  e  $y$  são as coordenadas espaciais e o valor de  $f$  na coordenada  $(x,y)$  fornece a intensidade da imagem no ponto. A Figura 1 é uma representação de uma imagem por meio de uma matriz de valores numéricos. Cada um desses valores se refere a um

ponto da imagem, que representa uma cor. Esses pontos são denominados *pixels*. Um *pixel* (abbreviatura para *picture element*) é a menor unidade de uma imagem, no domínio espacial, sobre a qual podem ser feitas operações [9]. A Figura 1 representa uma imagem por meio da matriz de *pixels* [2].

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0, n-1) \\ f(1,0) & f(1,1) & \dots & f(1, n-1) \\ \vdots & & & \\ f(m-1,0) & f(m-1,1) & \dots & f(m-1, n-1) \end{bmatrix}$$

**Figura 1** – Representação de uma imagem por meio da matriz de pixels

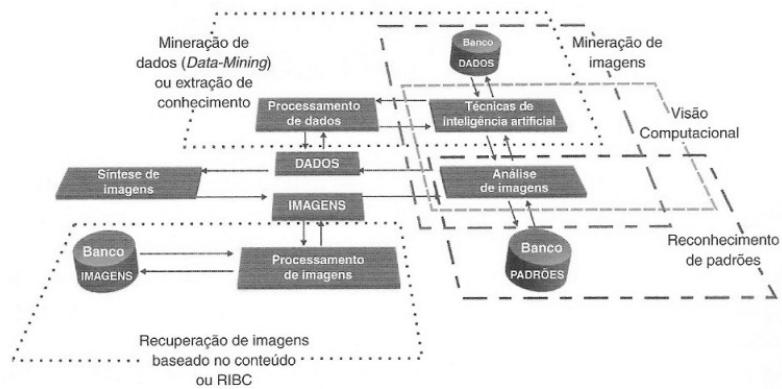
Segundo Nunes [2], o objetivo de definir matematicamente a imagem é a possibilidade de manipular o seu conteúdo a fim de realizar transformações ou extrair informações importantes. Intitula-se processamento de imagens todo o conjunto de operações que se pode aplicar em uma matriz que representa uma imagem, considerando-se o domínio espacial. Nesse sentido, o processamento de imagens pode ser categorizado em três níveis de processamento [8, 10], descrito na Tabela 1.

**Tabela 1** – Representação dos níveis de processamento de imagens digitais

Nível de Processamento	Área da computação gráfica
<b>Baixo nível</b>	<i>Pré-processamento de imagens:</i> esta área considera a manipulação das imagens depois de capturadas por dispositivos como satélites, <i>scanners</i> ou câmeras digitais. Sua finalidade é preparar as imagens para a identificação dos objetos no nível intermediário. Pode envolver o rearranjo de pontos (ou <i>pixels</i> ) e inclui tópicos como a diminuição de ruídos, realce de imagem, recuperação da imagem, dentre outros.
<b>Nível intermediário</b>	O nível intermediário de processamento de imagens atua em abstrações derivadas dos <i>pixels</i> das imagens e visa a auxiliar nas tomadas de decisões sobre as imagens em questão. A finalidade neste nível é segmentar a imagem, ou seja, dividir a imagem em partes significativas com o objetivo de facilitar o reconhecimento de padrões efetuado no nível alto.
<b>Alto nível</b>	<i>Análise das imagens/Visão computacional:</i> esta área trata da extração de informações das imagens, bem como a identificação e classificação de objetos presentes na imagem. Esta área pode envolver inteligência artificial (IA) ou o reconhecimento de padrões – que é o reconhecimento ou classificação de imagens considerando um catálogo de padrões possíveis ou um banco de padrões – para atingir algum grau de conhecimento.

Fontes: Conci et al. [8], Prajapati e Vij [10]

Atualmente as áreas relacionadas a sistemas de imagens digitais têm um campo de abrangência bem caracterizado. Se os dados são utilizados para a geração de imagens, a área de consideração é a síntese de imagens. Se esses dados forem resultados de informações adquiridas das imagens, a área considerada é a análise das imagens. O processamento de imagens é o ramo da computação gráfica que transforma as imagens, assim como o processamento de dados é o ramo da computação que transforma os dados [8]. A Figura 2 mostra a interação entre as diversas áreas relacionadas a sistemas de imagens digitais [8].



**Figura 2 – Representação da interação entre diversas áreas de sistemas de imagens digitais**

A capacidade humana de captar, processar e interpretar grandes volumes de dados representados visualmente estimula o desenvolvimento de metodologias cada vez mais sofisticadas. O processamento, a análise e a interpretação de imagens digitais são recursos importantes que visam a obter informações suficientes para avaliar diferentes objetos de interesse, de forma confiável e utilizando o mínimo de intervenção humana [1].

## 2.2 O modelo de cores RGB

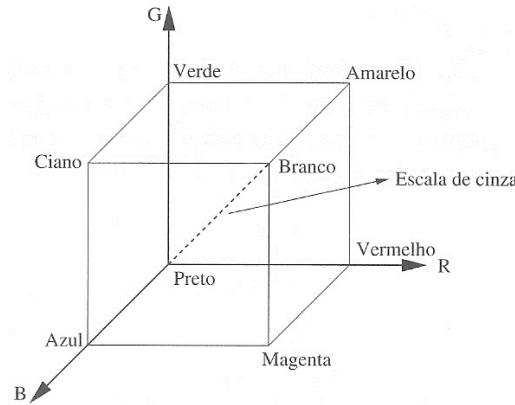
Esta seção tem como base o trabalho de Pedrini e Schwartz [1] e Burger e Burge [11].

As imagens coloridas são elementos importantes em várias atividades cotidianas das pessoas, pois estão presentes em áreas como a informática, a televisão, a fotografia e a impressão. A cor é uma propriedade importante na análise de imagens realizadas pelos seres humanos com ou sem a ajuda do computador.

Em sistemas de imagens digitais, os modelos de cores permitem a especificação em um formato padronizado para atender diferentes dispositivos gráficos ou aplicações que reque-

rem a manipulação de cores. Um modelo de cor é uma representação multidimensional na qual cada cor é especificada por um ponto no sistema de coordenadas multidimensionais.

O *Red-Green-Blue* (RGB) é um modelo de cores com base em um sistema de coordenadas cartesianas, em que o espaço de cores pode ser representado por um cubo, como ilustrado na Figura 3 [1].

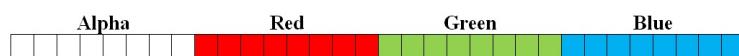


**Figura 3 – Representação do modelo RGB**

A Figura 3 mostra que as cores primárias vermelho (*Red*), verde (*Green*) e azul (*Blue*) estão em três vértices do cubo; as cores primárias complementares ciano, magenta e amarelo estão em outros três vértices; o vértice junto à origem representa a cor preta e o vértice mais afastado da origem corresponde à cor branca. A escala de cinza se estende através da reta que une a origem (preto) até o vértice mais distante (branco).

O RGB é um modelo aditivo, isto é, todas as cores partem do preto e são criadas por meio da adição de cores. Cada *pixel* da imagem é representado pelos três componentes (*Red*, *Green* e *Blue*) e, para criar cores diferentes, deve-se modificar a intensidade de cada componente independentemente. Diferentes intensidades de cada cor primária controlam o tom e o brilho da cor resultante. As cores cinza e branco são criadas por meio da associação das três cores primárias com a mesma intensidade ( $R=G=B$ ).

Na linguagem de programação Java, utilizada no desenvolvimento da presente pesquisa, é possível representar as imagens em RGB por meio de vetores de valores inteiros do tamanho de 32 bits. Como apresentado na Figura 4, 8 bits são utilizados para representar cada componente do RGB [11].

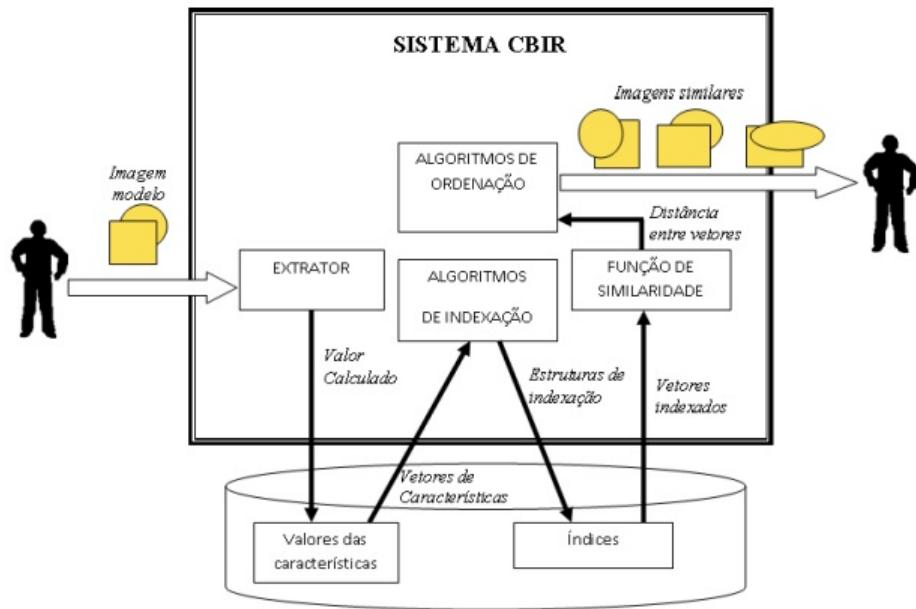


**Figura 4 – Representação de imagens no padrão RGB por meio de vetores**

A Figura 4 ilustra o vetor utilizado para representar as imagens RGB na linguagem de programação Java. Dentre os 32 bits possíveis, 8 são reservados para o valor *alpha* ( $\alpha$ ), que representa a transparência, ou seja, a habilidade de enxergar através da cor em relação ao plano de fundo. Esse tipo de representação limita a extensão de cada componente RGB aos valores entre 0 e 255, permitindo a representação de 16.777.216 cores distintas ( $256^3$ ).

## 2.3 Recuperação de imagens com base em conteúdo (CBIR)

Com o aumento das mídias digitais, o gerenciamento de informações se torna cada vez mais importante. Nas últimas décadas a técnica conhecida como CBIR (*Content-based Image Retrieval*), ou recuperação de imagens baseada em conteúdo, tem sido utilizada para a recuperação em grandes bases de imagens [12]. O sistema de CBIR utiliza conteúdos visuais, normalmente chamados de características, para encontrar imagens de acordo com as requisições do usuário a partir de uma imagem utilizada como modelo [13]. A Figura 5 apresenta a estrutura típica de um sistema de CBIR [14].



**Figura 5 – Estrutura típica de um sistema de CBIR**

Datta et al. [15] ressaltam que, em linhas gerais, o propósito de um sistema de CBIR não é encontrar uma imagem idêntica à imagem de consulta, o que poderia ser feito comparando-se todos os *pixels* de cada imagem. O propósito de tais sistemas é encontrar, em um banco de dados, as imagens mais similares em relação à imagem de consulta.

Um sistema de CBIR processa as informações contidas nas imagens digitais e cria uma abstração de tais imagens em termos de atributos visuais. No sistema de CBIR a similaridade entre duas imagens é calculada comparando as características extraídas por meio de técnicas de processamento de imagens. As características podem, por exemplo, se referirem a cor, textura ou forma e podem ser representadas por um vetor numérico que contém as informações sobre a imagem em questão. Tais características podem ser utilizadas individualmente ou em conjunto para a indexação e a recuperação de imagens [16].

### 2.3.1 Estágios do CBIR

Saha et al. [12] definem os principais estágios do CBIR como processamento, extração de características e indexação, a saber:

- **Processamento:** consiste em tratar as imagens visando a prepará-las para os estágios seguintes por meio da eliminação de ruídos, realce de contraste, isolamento de bordas, entre outros.
- **Extração de características:** consiste em extrair e armazenar as informações relevantes das imagens.
- **Indexação:** consiste em construir estruturas de dados para otimizar o processo de recuperação das imagens.

### 2.3.2 Extratores de características e funções de similaridade

Os algoritmos de extração de características são responsáveis por gerar valores numéricos que representam determinados aspectos de uma imagem. A qualidade dos extratores (também conhecidos como descritores) utilizados é muito importante em qualquer sistema de CBIR. Por isso, encontrar bons extratores para bases de dados genéricas pode ser uma tarefa muito difícil [16]. Em muitos casos, as características extraídas (e.g. cor, forma ou textura) são definidas por um vetor de alta dimensionalidade que representa diferentes níveis de detalhamento e permite melhor precisão na recuperação de imagens similares [17]. Muitos sistemas de CBIR utilizam mais de uma característica para encontrar as imagens similares, comparando o vetor obtido de uma imagem modelo com os demais vetores obtidos de outras imagens previamente processadas [12, 17].

Em um sistema de CBIR os vetores de características não são suficientes para determinar o resultado de uma consulta. Normalmente a consulta deve retornar as imagens mais similares, baseada em algum critério (ou função) de similaridade. Essas funções comparam os vetores de características e retornam as imagens mais próximas da imagem modelo, em termos de similaridade, medindo a distância entre os respecitivos vetores de características. Alguns exemplos de funções de distância bastante utilizados são: Euclidiana [18], *Minkowski* [12], *Hausdorff*, *Mahalanobis*, *Canberra*, *Cosseno*, divergência de Jeffrey e Qui-quadrado [19].

### 2.3.3 Estruturas de indexação

Em bases de dados pequenas e com poucas imagens é admissível a procura por imagens similares de maneira sequencial. Todavia, em grandes bases de dados, este tipo de consulta requer grande esforço computacional e demanda muito tempo, tornando-se impraticável. Com o objetivo de mitigar este problema surgiram as técnicas de indexação, que são essenciais para aumentar a eficiência da recuperação de imagens. Tais técnicas envolvem estudos de estruturas de dados e de banco de dados. Em geral, existem três abordagens utilizadas pelos algoritmos de indexação em sistemas de CBIR: 1) diminuir a dimensionalidade dos vetores de características; 2) organizar imagens semelhantes em grupos mais próximos no disco e 3) definir um espaço métrico, que considera a distância entre os elementos de um conjunto de dados para armazenar e recuperar imagens [18].

## 2.4 Teste de *software*

Muitas atividades de qualidade de *software* visam a encontrar soluções para mitigar ou até mesmo eliminar os erros e os riscos associados, pois análises e requisitos mal elaborados, problemas na execução do projeto e erros de codificação podem originar um *software* que não atenda aos critérios de qualidade.

O teste, considerado fundamental no cenário de desenvolvimento e de qualidade do *software*, é uma atividade que objetiva identificar possíveis defeitos o mais rápido possível [3]. Testar um *software* significa verificar, por meio de uma execução controlada, se o comportamento observado está de acordo com o comportamento especificado [5].

A definição dos dados que serão utilizados para testar o *software* representa um dos principais desafios inerentes à atividade de teste. Esses dados, que são os parâmetros de entrada utilizados durante a execução dos testes, são conhecidos como casos de teste

[6, 20] (vide seção 2.4.1). Em linhas gerais, os testes exaustivos (aqueles que englobam todas as possibilidades) são impraticáveis, pois um único caso de teste pode levar vários segundos/minutos para executar milhares de linhas de código, gerar centenas de processos concorrentes ou acessar diversos arquivos do sistema para operações de entrada/saída [21]. Portanto, devido às restrições de tempo e de recursos, os testes devem ser baseados em um subconjunto otimizado de casos de teste [22]. Aprimorar os testes inclui, dentre outras tarefas, projetar e selecionar casos de testes capazes de verificar tanto as condições válidas quanto as condições inválidas de execução do *software* [5].

Os critérios de teste representam os requisitos que o *software* deve atender para passar no teste [23], como por exemplo a quantidade de funções executadas com êxito ao término dos testes. De forma geral, tais critérios podem ser determinados a partir de três técnicas de teste. As técnicas funcionais (seção 2.4.2) utilizam as informações sobre as funcionalidades do *software* para definir os componentes a serem testados. As técnicas estruturais (seção 2.4.4) consideram a estrutura interna para estabelecer os elementos sob teste e as técnicas baseadas em defeitos (seção 2.4.3) são oriundas do conhecimento empírico sobre os erros tipicamente cometidos no processo de desenvolvimento [24].

#### 2.4.1 Casos de teste

Uma vez conhecidas as possíveis entradas e saídas do *software* sob teste, o próximo passo é definir os casos de teste. De acordo com o glossário de terminologias de Engenharia de Software do *Institute of Electrical and Electronics Engineers* (IEEE) [23], os casos de teste (CT) são parâmetros de entrada, condições de execução e resultados esperados utilizados para teste. A Figura 6 mostra um exemplo de casos de teste que podem ser utilizados para testar um *software* que emula uma calculadora.

Casos de teste para função de adição da calculadora		
Passo	Parâmetros de entrada	Resultado esperado
01	0+0	0
02	0+1	1
03	254+1	255
04	1028+1	1029

Fonte: adaptado de Patton [25].

**Figura 6** – Casos de teste: os diferentes parâmetros de entrada e os passos para se testar um *software*

A escolha dos parâmetros de entrada utilizados para testar o *software* é muito importante para a construção de estratégias de teste sólidas e realistas, pois as restrições

de tempo e de recursos devem ser consideradas. Tais parâmetros são conhecidos como “dados de teste” ou “dados de casos de teste” [25]. Apesar de sua definição mais ampla, o termo “casos de teste” também é muito utilizado pela comunidade de Engenharia de Software como referência direta aos “dados de teste”. Dessa forma, é comum intitular os métodos de geração de dados de teste, por exemplo, como “métodos de geração de casos de teste” ou métodos de seleção de dados de teste como “métodos de seleção de casos de teste”.

#### 2.4.2 Teste funcional

O teste funcional, também conhecido como teste de caixa-preta, é uma técnica para projetar casos de teste de acordo com as funcionalidades do *software*. Neste tipo de teste o sistema é avaliado segundo o ponto de vista do usuário e não se considera qualquer informação sobre sua implementação [3]. A execução desta técnica consiste em fornecer ao *software* os dados de entrada e, após o processamento, avaliar a saída e verificar se o resultado está em conformidade com as especificações. Alguns critérios mais conhecidos da técnica de teste funcional são:

- **Particionamento em classes de equivalência:** as classes de equivalência representam subdomínios de entrada que possuem funcionalidades similares. Por meio das especificações do *software*, divide-se o domínio de entrada em classes de equivalência válidas e inválidas. Com base na hipótese de que qualquer elemento de uma classe é representativo de toda a classe, seleciona-se um elemento de cada classe, de forma que cada novo caso de teste cubra o maior número de classes válidas possíveis. Tal procedimento reduz o domínio de entrada, porém deve ser capaz de representar todo o domínio de teste com algum nível de confiabilidade [3].
- **Análise do valor limite:** este critério é utilizado em conjunto com o particionamento em classes de equivalência, porém seu objetivo é explorar os limites inferiores e superiores de cada classe. Estudos empíricos mostram que os casos de teste que exploram condições limites têm uma maior probabilidade de encontrar defeitos [3].

#### 2.4.3 Teste baseado em defeitos

Os teste baseado em defeitos consiste na inclusão proposital de defeitos (considerados artificiais) no *software* sob teste como forma de projetar casos de teste que tenham uma alta probabilidade de encontrar defeitos [26]. Seu princípio básico é utilizar informações

sobre os tipos de erros mais frequentes no processo de desenvolvimento de *software* para derivar os requisitos de teste. Sua ênfase está nos erros que o programador pode cometer durante o desenvolvimento e nas abordagens que podem ser utilizadas para detectar a sua ocorrência [3].

#### 2.4.3.1 Semeadura de erros (*error seeding*)

Uma das formas mais comuns de análise do conjunto de testes se dá por meio de métodos estatísticos. É possível estimar, por exemplo, a quantidade de erros não descobertos por um determinado conjunto de teste utilizando técnicas de teste baseado em defeitos.

Mills [27] desenvolveu uma técnica para “semear” erros propositais no código e, a partir da inserção do erro proposital, estimar a quantidade de erros reais (não semeados) remanescentes no código. Assumindo-se que existe uma verossimilhança entre os erros semeados e os erros reais, executa-se os dados de testes para determinar a quantidade de erros semeados e de erros reais detectados. A partir da assunção de que os erros semeados e os erros reais possuem as mesmas propriedades estatísticas (e.g. possuem a mesma probabilidade de serem detectados) e que o erro semeado não é viciado, então a máxima verossimilhança é dada por

$$E = IS/K \quad (2.1)$$

Na equação 2.1  $I$  é a quantidade de erros reais detectados,  $S$  é a quantidade de erros semeados e  $K$  é a quantidade de erros semeados detectados. Essa estimativa, obviamente, pressupõe que os erros não detectados ( $E$ ), os erros reais detectados ( $I$ ) e os erros semeados ( $S$ ) são diretamente proporcionais.

#### 2.4.3.2 Testes de mutação (*mutation testing*)

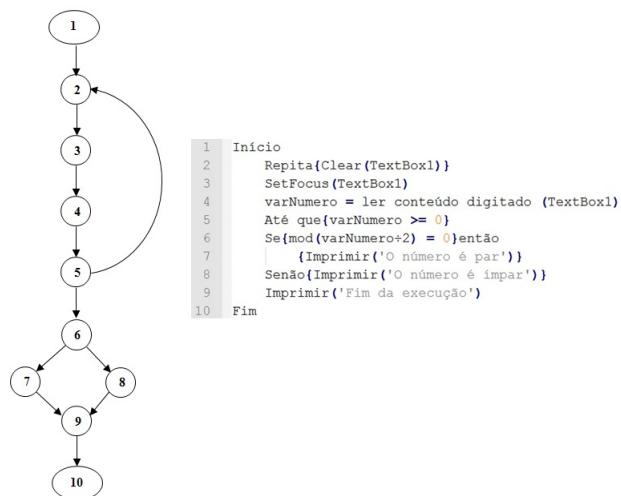
O teste de mutação é uma técnica baseada em defeitos e pode ser considerada um caso especial da semeadura de erros [28]. Os operadores de mutação modificam o *software* sob teste diversas vezes, produzindo um conjunto de programas alternativos, conhecidos como mutantes [3, 24]. Assim, um operador de mutação aplicado a um programa  $P$  o transforma em um programa similar  $P'$ . A produção de tais erros estimula o testador a construir e selecionar casos de teste que, por meio a análise dos mutantes, revelem que tais modificações conduzem a um funcionamento incorreto do programa [3].

A intenção do teste de mutação é avaliar a eficácia dos casos de teste. Após a execução do teste de mutação, considera-se um bom caso de teste aquele que produz um resultado diferente entre os mutantes e o *software* sob teste, pois, teoricamente, os casos de teste capazes de revelar defeitos nos mutantes (que certamente contêm erros) devem ser capazes de revelar eventuais defeitos no programa original. Isso é conhecido como “efeito de acoplamento” [3].

#### 2.4.4 Teste estrutural

O teste estrutural, também conhecido como teste de caixa-branca, é uma abordagem para projetar casos de teste derivados da estrutura e da implementação do *software*. Informalmente se pode definir tal abordagem como uma atividade na qual são testados os caminhos lógicos do *software* por meio de casos de teste capazes de executar condições específicas e/ou exercitar as estruturas de dados internas pelo menos uma vez, objetivando garantir a sua validade [3].

Nos testes de caixa-branca, é comum utilizar uma notação conhecida como grafo de fluxo de controle (GFC) para representar os caminhos de execução do *software* sob teste. A ideia principal é que um programa pode ser decomposto em blocos disjuntos de comandos. A execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, de forma sequencial. A partir do GFC se pode escolher os elementos que devem ser testados, caracterizando o teste estrutural. A Figura 7 é uma representação de um grafo de fluxo de controle (incluindo seu pseudocódigo).



Fonte: adaptado de Delamaro et al. [3].

**Figura 7** – Representação simplificada de um grafo de fluxo de controle

Na Figura 7, os blocos de comandos 2–5 são uma representação da estrutura de comandos “Repita...Até que”, na qual o *software* só continua a sua execução quando o usuário inserir um número igual ou maior que zero. Já os blocos 6–7 e 6–8 são mutuamente exclusivos e representam uma estrutura de comandos do tipo “Se...então...Senão”, na qual o *software* verifica se o número inserido é par ou ímpar, imprimindo em tela a informação relacionada ao resultado da comparação. Por fim, o bloco 9 imprime em tela o aviso de fim da execução.

#### 2.4.5 Oráculos de teste

A automatização dos testes é um processo que visa a mitigar ou eliminar os testes manuais e subjetivos, além de contribuir para o aumento de produtividade e para a consequente otimização dos recursos e dos custos. Entretanto, as ferramentas de geração, de seleção e de priorização de casos de teste, por si sós, podem não ser suficientes para garantir a eficácia deste processo, pois existe a necessidade de se determinar se os resultados (saídas) atendem ou não aos requisitos de teste [29].

Os oráculos de teste, ou simplesmente oráculos, são mecanismos utilizados para definir a saída ou o comportamento esperado de uma execução do *software* sob teste [30]. O papel de um oráculo pode ser desenvolvido por humanos ou por ferramentas automatizadas de teste. Os oráculos podem ser, por exemplo, as especificações ou o conhecimento prévio do *software* sob teste [17]. Já um oráculo automatizado pode ser constituído por um protótipo, por uma versão mais antiga ou pela saída de *software* que se admite estar correta [18, 31].

A automatização dos oráculos não é uma tarefa trivial. Um *software* pode ter várias configurações possíveis e pode trabalhar com diferentes tipos de dados (e.g. textos, requisições *web*, dados multimídia, entre outros) e, quanto mais complexo for o *software*, mais complexa pode ser a definição e a automatização de um oráculo de teste [17, 18].

Para auxiliar na automatização de oráculos para *software* em que o domínio de saída sejam representados por imagens, Oliveira et al. [17] desenvolveram o *framework* *O-FIm*, cujo objetivo é permitir a comparação entre duas imagens por meio da utilização de extractores de características e de funções de similaridade. Os detalhes do *O-FIm* serão discutidos na próxima seção.

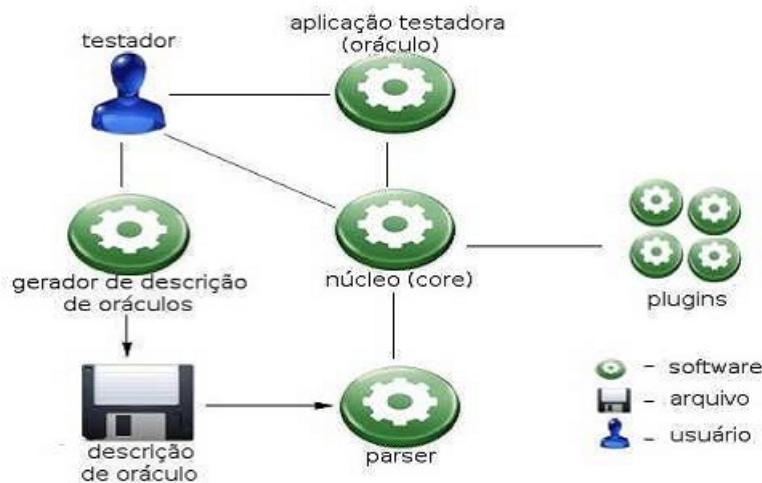
## 2.5 O framework *O-FIm* (*Oracle for Images*)

Na programação orientada a objetos (POO), um *framework* pode ser entendido como um conjunto de classes que suporta o reuso em diferentes níveis de especificações [32, 33]. Ele representa um modelo abstrato desenvolvido para uma finalidade específica e permite não apenas o reuso de código, mas também o reuso do modelo. A estrutura do *framework* abstrato pode ser completada com as bibliotecas da própria aplicação, provendo flexibilidade ao desenvolvedor para adaptar as funcionalidades desejadas.

Delamaro et al. [18] apresentaram um *framework* para definição de oráculos de teste para *software* com saídas gráficas intitulado *O-FIm* (*Oracle for Images*). O *O-FIm* é uma ferramenta desenvolvida na linguagem Java que permite ao testador comparar duas imagens previamente armazenadas e obter um resultado que indica o nível de similaridade entre as imagens. Tal ferramenta foi desenvolvida para programadores/testadores, portanto exige de seus usuários conhecimentos da linguagem Java e de processamento de imagens para sua correta implementação. O *O-FIm* fornece flexibilidade e facilidade de uso, pois permite ao testador escolher e configurar as características das imagens que serão utilizadas para comparação, bem como instalar extratores e funções de similaridade de maneira simples [17]. O *O-FIm* possui vários extratores de características e várias funções de similaridade que podem ser reutilizadas [34], dentre os quais destacam-se:

1. **Extratores:** extrator de cor, que retorna a média de cores da imagem; extrator de área, que conta a quantidade de *pixels* dentro da borda da imagem; extrator de perímetro, que sumariza os *pixels* que fazem parte da borda demarcada e extrator de assinatura, que retorna a informação sobre o formato da borda das imagens.
2. **Funções de similaridade:** distância Euclidiana; distância *Minkowski*; distância de *Hausdorff*; distância *Mahalanobis*; distância *Canberra*; distância Cosseno; divergência de *Jeffrey* e Qui-quadrado.

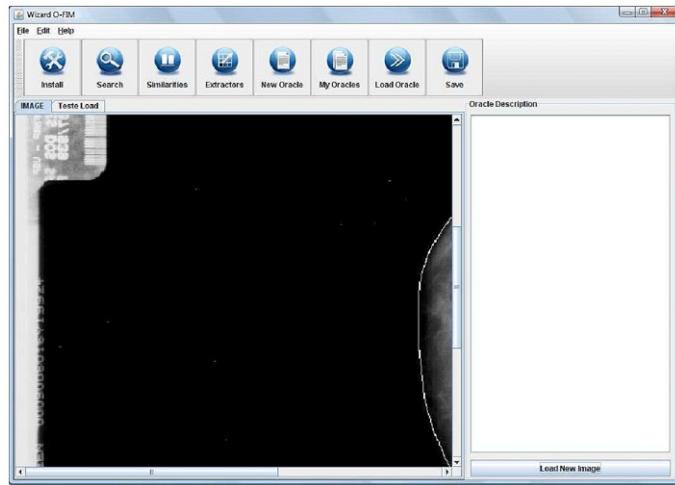
A portabilidade e as diversas bibliotecas da linguagem Java viabilizam a configuração de um ambiente flexível e possibilita a instalação de extratores de características e funções de similaridade para gerar oráculos de teste aplicáveis em programas com saídas gráficas. A Figura 8 ilustra a arquitetura do *O-FIm*, a qual envolve os seus componentes: núcleo, descrição dos oráculos gráficos e *plugins* [18].



**Figura 8 – Arquitetura do framework *O-FIm***

- **Núcleo:** o núcleo do *O-FIm* permite a instalação e remoção de *plugins* e disponibiliza uma *Application Programming Interface* (API) a partir da qual o testador pode construir seus oráculos.
- **Descrição dos oráculos gráficos:** o *O-FIm* depende de uma descrição textual feita em arquivo texto por meio da qual o testador pode exprimir suas especificações particulares para o teste.
- **Plugins:** para que a adaptação de conceitos de CBIR permita o correto funcionamento do *O-FIm* é necessário que qualquer classe Java (e.g. extratores ou funções de similaridade) implemente as *interfaces* denominadas “*IExtractor*”, para extratores, e “*ISimilarity*”, para funções de similaridade. Ao implementar tais *interfaces*, o *plugin* indica para a ferramenta como identificar, localizar e utilizar suas classes no ambiente do oráculo gráfico.

Em suma, o *O-FIm* emprega conceitos de CBIR para a automatização de oráculos para programas cuja saída é apresentada na forma de uma imagem, seja uma interface gráfica, seja qualquer imagem produzida pelo *software* sob teste. A Figura 9 apresenta a interface principal do *O-FIm* [18].



**Figura 9** – Interface principal do framework O-FIm

Na interface apresentada na Figura 9, o testador pode instalar, procurar e remover *plugins*. Também é possível selecionar os extratores desejados e as funções de similaridade, além de avaliar as suas propriedades.

## 2.6 Seleção de casos de teste

A seleção de casos de teste é uma abordagem que objetiva escolher um subconjunto de dados de teste dentro de um domínio específico de acordo com algum critério de interesse. Normalmente tal abordagem considera as condições de execução e os resultados esperados para testar o *software*. A seleção é muito importante para a definição e a construção de estratégias de teste, pois visa a eliminar os dados de teste redundantes ou desnecessários e a maximizar a detecção de falhas [35]. A problemática de casos de teste pode ser categorizada em quatro aspectos fundamentais: geração, minimização, seleção e priorização de casos de teste.

Os geradores de casos de teste são programas que recebem como parâmetros de entrada o código fonte, os critérios de teste, as especificações ou as definições de estrutura de dados e utiliza esses parâmetros para gerar dados de teste [23].

Os algoritmos de geração de casos de teste utilizam heurísticas ou estratégias específicas visando a criar casos de teste capazes de maximizar a cobertura dos testes. Entretanto, gerar casos de teste realistas e que atendam aos requisitos de teste não é uma tarefa trivial, haja vista a complexidade e a diversidade dos parâmetros de entrada de aplicações do mundo real.

Na prática, um enorme conjunto de testes pode ser criado utilizando ferramentas como a Tobias [36] ou a TRUST [37]. Contudo, a execução de um grande conjunto de testes é, no mínimo, indesejável. Os requisitos de tempo e de recursos devem ser considerados e, nesses casos, há a necessidade de se aplicar métodos capazes de reduzir a quantidade de casos de teste a serem utilizados.

A redução do conjunto de testes objetiva eliminar os casos de teste redundantes ou desnecessários e consequentemente diminuir a quantidade de dados utilizados para teste. Existem duas principais alternativas para reduzir o conjunto de testes: a minimização de casos de teste (também conhecida como minimização do conjunto de teste) e a seleção de casos de teste. A minimização consiste em gerar um número otimizado de casos de teste por meio da contenção de critérios de teste, como por exemplo a contenção de cobertura de código ou de modelo [38]. Uma abordagem muito utilizada para a minimização é o particionamento do conjunto de testes. Esse método consiste em selecionar um subconjunto otimizado de elementos dentre os casos de teste válidos, conforme ilustrado na Figura 10.



**Figura 10** – Particionamento do conjunto de testes

A Figura 10 representa o particionamento do conjunto de testes. Neste exemplo,  $CT_{RO}$  representa os casos de teste redundantes ou obsoletos pertencentes ao conjunto de testes;  $CT_V$  representa os casos de teste válidos e  $CT_{OV}$  representa os casos de testes válidos e otimizados. É importante destacar que  $CT_{OV}$  é um subconjunto de todos os casos de teste válidos.

Uma forma de implementar o particionamento do conjunto de testes é considerar apenas os casos de teste relacionados com as mudanças de configurações mais relevantes do *software* sob teste, em sistemas que possuem um grande número de configurações possíveis, conforme apresentado por Qu et al. [39]. Zeng et al. [40] utilizaram a abordagem “*attribute relevance analysis*”, na qual o particionamento é realizado de acordo com o grau de relevância do atributo, conferido por meio da classificação e análise de correlação dos

dados. Ensan et al. [41] propuseram uma estratégia para redução do conjunto de testes no paradigma de linhas de produtos de *software* (*software product lines*), utilizado para modelar todas as configurações de um *software* dentro de um determinado domínio. Nessa abordagem, as características e as configurações são priorizadas e selecionadas de acordo com seu grau de importância, proporcionando a redução do conjunto de testes.

A seleção de casos de teste normalmente é tratada como uma abordagem que pode abranger dois principais paradigmas: 1) em testes de regressão<sup>†</sup>, a seleção é centrada nas modificações do *software* sob teste (SUT). Nesse contexto, os casos de teste são escolhidos porque são relevantes para as partes alteradas do SUT. Esse processo comumente envolve uma análise estática de caixa-branca e do código-fonte [38] e 2) seleção de casos de teste utilizando conceitos de similaridade. Nesse tipo de abordagem, a seleção se fundamenta na heurística de que os casos de teste mais dissimilares são preferíveis para a detecção de falhas. Para Hemmati et al. [35] o objetivo da seleção com base em similaridade é escolher um subconjunto de elementos que maximize a capacidade de detecção de falhas, a partir de um conjunto que contém uma quantidade máxima de casos de testes possíveis.

Algumas definições sutis, porém importantes, podem ser consideradas para distinguir abordagens de minimização e de seleção de casos de teste:

- diferentemente da seleção, a minimização normalmente produz uma diminuição permanente do conjunto de testes [38];
- a minimização não considera as partes do *software* que foram modificadas [38] e
- a minimização por meio da contenção de critérios de teste, em geral, não permite escolher uma quantidade predefinida de casos de teste disponíveis para execução, o que pode representar uma desvantagem em relação à seleção de casos de teste [42].

Outra abordagem importante para a definição dos casos de teste é a priorização de casos de teste. A priorização ordena os casos de teste sequencialmente e permite que os dados de teste mais importantes sejam executados primeiro. Seu objetivo é encontrar a presença de falhas o mais cedo possível e permitir uma depuração mais rápida do SUT. A priorização geralmente foca o histórico de execução dos casos de teste e os impactos das mudanças realizadas no *software* [43].

Em suma, o foco da geração de casos de teste é utilizar parâmetros de entrada para gerar dados de teste. A minimização do conjunto de testes normalmente consiste em gerar

---

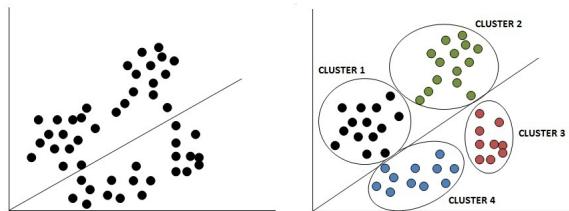
<sup>†</sup>Os testes de regressão se aplicam a uma nova versão de *software*, visando a revalidá-lo após sua modificação [3].

um número reduzido de casos de teste por meio da contenção de critérios de teste ou pelo particionamento do conjunto de teste. A seleção procura escolher um subconjunto de casos de teste a partir de todo o conjunto de testes utilizando algum critério de interesse e a priorização ordena os casos de teste de acordo com sua relevância. Em linhas gerais, existem algoritmos de geração, minimização e priorização que utilizam métodos similares aos empregados em SCT, porém as abordagens e os objetivos são diferentes.

## 2.7 Clustering

*Clustering* é uma técnica utilizada para agrupar objetos de acordo com seu grau de similaridade, minimizando a variância entre os elementos do mesmo grupo e maximizando a variância entre os elementos de grupos distintos [44].

O critério de similaridade utilizado no *clustering* pode ser, por exemplo, a variabilidade entre dois ou mais objetos pertencentes ao conjunto de dados, como a média, desvio-padrão, mediana, covariância, entre outros. O objetivo do *clustering* é criar *clusters* heterogêneos com conteúdo homogêneo, de forma que cada *cluster* possui objetos que são similares entre si, porém dissimilares dos objetos dos outros *clusters* [6]. A Figura 11 é uma representação diagramática do processo de *clustering*.



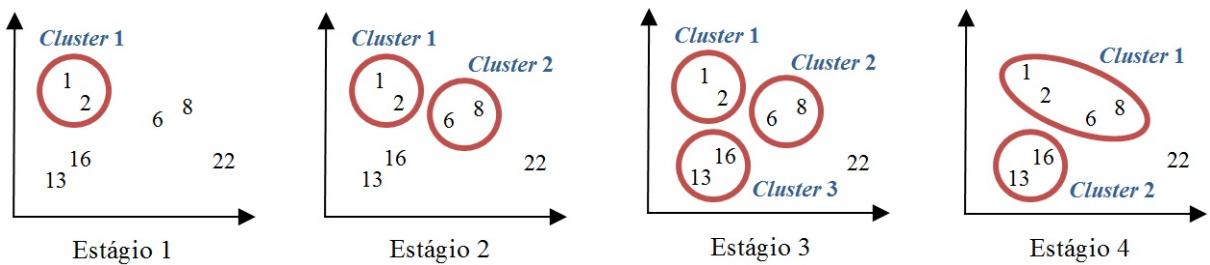
Fonte: adaptado de Sapna e Mohanty [6].

**Figura 11** – Representação diagramática do processo de *clustering*

A obtenção de grupos condensados de dados tende a aumentar o poder de inferência sobre o conjunto de dados observados. A análise de agrupamentos (*cluster analysis*) é um termo usado para descrever diversos métodos cujo propósito é classificar os dados sob estudo em grupos discretos. Os métodos mais tradicionais de *clustering* são os métodos hierárquicos e métodos por particionamento [45]. As próximas subseções apresentam dois métodos clássicos de *clustering*: o método hierárquico *Agglomerative Hierarchical Clustering* e o método por particionamento *K-Means*.

### 2.7.1 Método aglomerativo hierárquico (AHC)

No método AHC (*Agglomerative Hierarchical Clustering*) a similaridade entre dois *clusters* é definida pelos dois elementos mais semelhantes entre si. Inicialmente, cada elemento (dado) representa um *cluster* isolado. A cada iteração os dois *clusters* mais similares são agrupados em um único *cluster*. A Figura 12 representa os primeiros estágios do processo de AHC.



**Figura 12 – Representação do processo de AHC**

A Figura 12 mostra que, nos três primeiros estágios desse exemplo, os *clusters* mais próximos (ou mais similares) são  $(1) \leftrightarrow (2)$ ,  $(6) \leftrightarrow (8)$  e  $(13) \leftrightarrow (16)$ , respectivamente. Nesses estágios cada *cluster* isolado é agrupado gerando novos *clusters* (*cluster 1*, *cluster 2* e *cluster 3*). É importante destacar que a similaridade é medida pela menor distância entre cada par de *clusters*. No quarto estágio os *clusters* mais próximos são  $(1, 2) \leftrightarrow (6, 8)$ . Nesse estágio, os *clusters* são agrupados e um novo *cluster* é gerado  $(1, 2, 6, 8)$ . O processo se repete até atingir o ponto de parada desejado ou até que todos os dados sejam agrupados em um único *cluster*. Para exemplificar o método AHC aplicar-se-á tal método ao conjunto de dados da Tabela 2 utilizando a distância Euclidiana para a comparação dos grupos.

**Tabela 2 – Renda e idade de seis indivíduos distintos**

Indivíduo	Renda	Idade
A	9,60	28
B	8,40	31
C	2,40	42
D	18,20	38
E	3,90	25
F	6,40	41
Média	8,15	34,17
Desvio Padrão	5,61	7,14

Fonte: Mingoti [46].

**Passo 1.** A matriz de distâncias D entre os seis elementos amostrais é dada por:

	A	B	C	D	E	F
A	0	3,23	15,74	13,19	6,44	13,39
B	3,23	0	12,53	12,04	7,50	10,19
C	15,74	12,53	0	16,29	17,06	4,12
D	13,19	12,04	16,29	0	19,33	12,18
E	6,44	7,50	17,06	19,33	0	16,19
F	13,39	10,19	4,12	12,18	16,19	0

É importante notar que a matriz D é simétrica. Os pontos redundantes e os pontos cuja distância é igual a zero aparecem em destaque (cor verde) na matriz.

O menor valor observado na matriz D é 3,23 (também chamado de nível de fusão), que corresponde à distância entre os elementos A e B nas duas variáveis medidas. Esses indivíduos são, então, reunidos em um único *cluster*, e a amostra de seis elementos é repartida em cinco *clusters* que são:

$$C_1 = \{A, B\}, C_2 = \{C\}, C_3 = \{D\}, C_4 = \{E\} \text{ e } C_5 = \{F\}.$$

Nesse passo do algoritmo, os valores de distância obtidos entre os *clusters* {A, B} em relação aos outros é:

$$d(\{A, B\}, \{C\}) = \min \{d(A, C); d(B, C)\} = \min \{15,74; 12,53\} = 12,53.$$

$$d(\{A, B\}, \{D\}) = \min \{d(A, D); d(B, D)\} = \min \{13,19; 12,04\} = 12,04.$$

$$d(\{A, B\}, \{E\}) = \min \{d(A, E); d(B, E)\} = \min \{6,44; 7,50\} = 6,44.$$

$$d(\{A, B\}, \{F\}) = \min \{d(A, F); d(B, F)\} = \min \{13,39; 10,19\} = 10,19.$$

	A	B	C	D	E	F
A	0	3,23	apagado	apagado	6,44	apagado
B	3,23	0	12,53	12,04	apagado	10,19
C	apagado	12,53	0	16,29	17,06	4,12
D	apagado	12,04	16,29	0	19,33	12,18
E	6,44	apagado	17,06	19,33	0	16,19
F	apagado	10,19	4,12	12,18	16,19	0

**Passo 2.** A matriz de distâncias entre os cinco grupos formados no passo 1 é dada por:

$$D_{5 \times 5} = \begin{bmatrix} & \{A, B\} & C & D & E & F \\ \{A, B\} & 0 & 12, 53 & 12, 04 & 6, 44 & 10, 19 \\ C & 12, 53 & 0 & 16, 29 & 17, 06 & 4, 12 \\ D & 12, 04 & 16, 29 & 0 & 19, 33 & 12, 18 \\ E & 6, 44 & 17, 06 & 19, 33 & 0 & 16, 19 \\ F & 10, 19 & 4, 12 & 12, 18 & 16, 19 & 0 \end{bmatrix}$$

O valor mínimo da matriz D agora é 4,12, que é a distância entre os indivíduos C e F. Portanto, a amostra fica dividida em quatro *clusters* que são:

$$C_1 = \{A, B\}, C_2 = \{C, F\}, C_3 = \{D\} \text{ e } C_4 = \{E\}.$$

$$D_{5 \times 5} = \begin{bmatrix} & \{A, B\} & C & D & E & F \\ \{A, B\} & 0 & \text{apagado} & 12, 04 & 6, 44 & 10, 19 \\ C & \text{apagado} & 0 & \text{apagado} & \text{apagado} & 4, 12 \\ D & 12, 04 & \text{apagado} & 0 & 19, 33 & 12, 18 \\ E & 6, 44 & \text{apagado} & 19, 33 & 0 & 16, 19 \\ F & 10, 19 & 4, 12 & 12, 18 & 16, 19 & 0 \end{bmatrix}$$

**Passo 3.** A matriz de distâncias é dada por:

$$D_{4 \times 4} = \begin{bmatrix} & \{A, B\} & \{C, F\} & D & E \\ \{A, B\} & 0 & 10, 19 & 12, 04 & 6, 44 \\ \{C, F\} & 10, 19 & 0 & 12, 18 & 16, 19 \\ D & 12, 04 & 12, 18 & 0 & 19, 33 \\ E & 6, 44 & 16, 19 & 19, 33 & 0 \end{bmatrix}$$

O valor mínimo da matriz D é 6,44, que é a distância entre  $\{A, B\}$  e E. Portanto, estes dois *clusters* são combinados num único *cluster*, e a amostra fica repartida em três *clusters*:

$$C_1 = \{A, B, E\}, C_2 = \{C, F\} \text{ e } C_3 = \{D\}.$$

$$D_{4 \times 4} = \begin{bmatrix} & \{A, B\} & \{C, F\} & D & E \\ \{A, B\} & 0 & 10, 19 & 12, 04 & 6, 44 \\ \{C, F\} & 10, 19 & 0 & 12, 18 & \text{apagado} \\ D & 12, 04 & 12, 18 & 0 & \text{apagado} \\ E & 6, 44 & \text{apagado} & \text{apagado} & 0 \end{bmatrix}$$

**Passo 4.** A matriz de distâncias é dada por:

$$D_{3 \times 3} = \begin{bmatrix} & \{A, B, E\} & \{C, F\} & D \\ \{A, B, E\} & 0 & 10, 19 & 12, 04 \\ \{C, F\} & 10, 19 & 0 & 12, 18 \\ D & 12, 04 & 12, 18 & 0 \end{bmatrix}$$

O valor mínimo é 10,19, que corresponde à junção dos *clusters* {A, B, E} e {C, F}. Assim a amostra fica repartida em dois grupos:

$$C_1 = \{A, B, E, C, F\} \text{ e } C_2 = \{D\}.$$

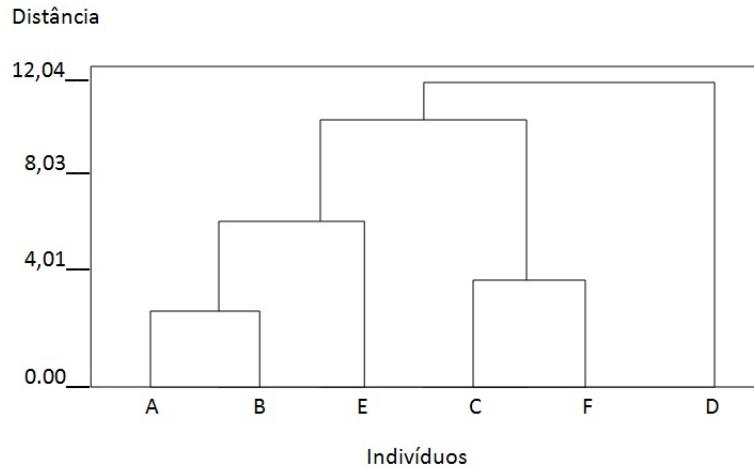
**Passo 5.** Este é o último passo do algoritmo de agrupamento, e a amostra se reduz a um único *cluster* que é  $C_1 = \{A, B, C, D, E, F\}$ , sendo o nível de fusão (menor distância entre os *clusters*) final igual a 12,04.

A Tabela 3 apresenta o resumo dos resultados obtidos após a execução do algoritmo de AHC. Tais resultados permitem a construção de um dendograma para representação gráfica do processo de *clustering*, apresentado na Figura 13.

**Tabela 3 – Resultados obtidos após a execução do algoritmo AHC**

Passo	# Grupos	Fusão	Distância (nível de fusão)
1	5	{A} e {B}	3,23
2	4	{C} e {F}	4,12
3	3	{A,B} e {F}	6,44
4	2	{A,B,E} e {C,F}	10,19
5	1	{A,B,E,C,F} e {D}	12,04

Fonte: Mingoti [46].



Fonte: Mingoti [46].

**Figura 13 – Dendograma do processo de clustering, método AHC**

Uma questão de suma importância no AHC é determinar em qual passo o algoritmo de *clustering* deve ser interrompido, bem como definir a quantidade de *clusters* da partição final do conjunto de dados analisados. Essa avaliação normalmente é subjetiva e depende do contexto de implementação do método de *clustering*. Entretanto, existem algumas

métricas que podem auxiliar na decisão final, como a análise do comportamento do nível de fusão e a análise do comportamento do nível de similaridade:

- **Análise do comportamento do nível de fusão (distância):** conforme as iterações do algoritmo de *clustering*, a similaridade entre os elementos que estão sendo combinados vai decrescendo e, consequentemente, a distância entre eles vai aumentando. Dessa maneira, é possível analisar se há pontos de salto relativamente grandes em relação aos demais valores de distância. Esses pontos podem indicar o momento ideal para a parada do algoritmo. Logo, se a função apresentar vários pontos de salto é possível determinar uma região de prováveis valores para a quantidade de *clusters* que melhor classificam os dados analisados. O dendograma também pode ser utilizado como ferramenta para a visualização dos pontos de salto dos níveis de similaridade.
- **Análise do comportamento do nível de similaridade:** este critério é similar ao critério descrito anteriormente, porém o que se observa é o nível de similaridade entre os elementos. Se  $C_i$  e  $C_l$  são os *clusters* unidos num determinado estágio  $E$ , o nível de similaridade é definido pela equação 2.2.

$$NS(C_i, C_l) = \left( 1 - \frac{distancia(C_i, C_l)}{\max\{distancia_{j,k}, j, k = 1, 2, \dots, n\}} \right) \quad (2.2)$$

No qual  $\max\{distancia_{j,k}, j, k = 1, 2, \dots, n\}$  é a maior distância entre os  $n$  elementos amostrais na matriz  $D_{n \times n}$  no primeiro estágio do processo de *clustering*. Nesse caso, procura-se detectar pontos nos quais há um decrescimento acentuado na similaridade dos elementos unidos, pontos estes que podem indicar que o algoritmo deve ser interrompido.

O exemplo a seguir ilustra o cálculo do nível de similaridade. A maior distância entre os  $n$  elementos da matriz de similaridade  $D$  é 19,33.

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
<i>A</i>	0	3,23	15,74	13,19	6,44	13,39
<i>B</i>	3,23	0	12,53	12,04	7,50	10,19
<i>C</i>	15,74	12,53	0	16,29	17,06	4,12
<i>D</i>	13,19	12,04	16,29	0	19,33	12,18
<i>E</i>	6,44	7,50	17,06	19,33	0	16,19
<i>F</i>	13,39	10,19	4,12	12,18	16,19	0

No primeiro estágio do processo de *clustering*, o nível de similaridade é definido por:

$$NS_{A,B} = \left( 1 - \frac{3,23}{19,33} \right) = 0,83. \quad (2.3)$$

E no segundo estágio do processo de *clustering*, o nível de similaridade é definido por:

$$NS_{C,F} = \left( 1 - \frac{4,12}{19,33} \right) = 0,79. \quad (2.4)$$

### 2.7.2 Método *K-Means*

O *K-Means*<sup>†</sup> é um método de particionamento muito popular na tarefa de *clustering*. No primeiro passo o algoritmo toma um conjunto de dados de entrada e escolhe, aleatoriamente,  $K$  pontos de dados como sendo os elementos centrais de cada *cluster*. Cada um desses pontos é conhecido como centróide, ou centro de gravidade do *cluster*. Em seguida, cada um dos pontos remanescentes é atribuído ao *cluster* mais similar, com base na distância entre o ponto e o centróide.

No segundo passo o algoritmo calcula a média com base em todos os pontos do *cluster*. A média do *cluster* passa a ser o centróide e novamente cada um dos pontos remanescentes é atribuído ao *cluster* mais similar, conforme a distância entre o ponto e o centróide. O processo se repete até que os centróides parem de se modificar (atingindo a convergência) ou após um número limitado de iterações previamente especificado. A similaridade em um *cluster* é medida em relação ao valor médio dos pontos contidos nesse *cluster* (centróide). O pseudocódigo do método *K-Means* é apresentado no Algoritmo 1.

<b>Entrada:</b> Conjunto de dados $D$ <b>1</b> Escolher $K$ centróides aleatoriamente <b>2</b> enquanto <i>Houver mudança no valor dos centróides</i> faça <b>3</b> <b>para cada ponto</b> $p$ de $D$ <b>hacer</b> <b>4</b> Calcular a distância entre $p$ e os centróides <b>5</b> Encontrar a menor distância <b>6</b> Atribuir $p$ ao <i>cluster</i> <b>7</b> Calcular a média dos <i>clusters</i> <b>8</b> Atualizar os centróides <b>9</b> <b>retorna</b> Apresentar <i>clusters</i>
--

**Algoritmo 1:** *K-Means*

---

<sup>†</sup>O conteúdo sobre o método *K-Means* tem como base principal o trabalho de Goldschmidt e Passos [45].

O método *K-Means* é relativamente escalável e eficiente no processamento de grandes conjuntos de dados, pois a complexidade computacional do algoritmo é  $(n * k * t)$ , onde  $n$  é o número total de objetos,  $k$  é o número de *clusters*, e  $t$  é o número de iterações. É desejável que  $k \ll n$  e  $t \ll n$ .

Em aplicações reais frequentemente existem dados que não seguem o comportamento geral do conjunto de dados remanescentes e são conhecidos como ruídos (*outliers*). Os ruídos podem existir, por exemplo, devido à variabilidade dos dados, como quando um objeto tem um valor muito discrepante em relação aos demais, mas trata-se de um valor correto (verdadeiro). O *K-Means* é sensível a ruídos, visto que pequeno número desses dados pode influenciar substancialmente o valor médio. O algoritmo também é sensível à partição inicial, gerada pela escolha aleatória dos centróides.

Outra desvantagem potencial é a necessidade do usuário especificar uma quantidade prévia de *K clusters*. Em geral, diversos experimentos empíricos variando o valor de *K* devem ser realizados.

## 2.8 Considerações finais

Esse capítulo abordou os principais conceitos que formam a base teórica para a estruturação e a implementação do método apresentado no capítulo 4. Para atingir o objetivo proposto é necessário o emprego de diferentes métodos computacionais. As técnicas de *Content-Based Image Retrieval* permitem a abstração e a extração de características das imagens, que podem ser armazenadas em um banco de dados para posterior manipulação. O emprego de *frameworks* torna possível o reuso de *software* em alto nível, enquanto os métodos de *clustering* permitem classificar e aumentar o poder de inferência sobre os dados. Por fim, a definição e a organização da massa de dados permite a aplicação de métodos de seleção de casos de teste, atividade fundamental no processo de teste de *software*.

O próximo capítulo apresenta uma revisão bibliográfica sistemática sobre seleção de casos de teste. Seu propósito é aferir o estado da arte neste domínio.

# *Capítulo 3*

## *Revisão Bibliográfica Sistemática*

Esse capítulo apresenta uma revisão sistemática (RS) sobre seleção de casos de teste e visa a propiciar um levantamento bibliográfico preciso e auditável, bem como aferir o estado da arte por meio da extração e análise de dados oriundos de fontes científicas na área de Computação. As principais abordagens analisadas foram classificadas de acordo com os objetivos de teste originalmente reportados em cada artigo considerado, e incluem as heurísticas, o domínio de aplicação e as métricas de avaliação dos métodos de seleção de casos de teste.

A revisão sistemática é uma metodologia rigorosa e confiável, cujo objetivo é identificar e selecionar pesquisas relevantes, coletar e analisar os dados obtidos e permitir a auditoria [49]. Esse trabalho foi conduzido considerando as três fases sugeridas por Kitchenham [50]: planejamento, condução e extração de dados. As seguintes terminologias pesquisadas “geração, seleção e priorização de casos de teste”, “critérios e objetivos de teste”, “conjunto de teste” bem como “requisitos e cobertura de teste” estão de acordo com as especificações do IEEE [23].

Para atingir o objetivo proposto o presente capítulo está estruturado como segue: seção 3.1 apresentando o planejamento (seleção de trabalhos); na seção 3.2 é apresentada a condução da revisão sistemática; a seção 3.3 mostra os resultados e as discussões pertinentes à revisão sistemática; a seção 3.4 mostra os trabalhos relacionados; na seção 3.5 são apresentadas as tendências e os desafios relacionados às pesquisas em seleção de casos de teste; na seção 3.6 são apresentadas as limitações da pesquisa; a seção 3.7 apresenta as conclusões e os trabalhos futuros e, finalmente, a seção 3.8 apresenta as considerações finais.

### **3.1 Seleção de trabalhos**

Os critérios de inclusão ou exclusão da revisão sistemática objetivam delimitar a seleção de trabalhos com base em avaliações qualitativas relevantes ao objeto de pesquisa. Na presente RS os trabalhos incluídos deveriam apresentar propostas, estudos ou experi-

mentos considerando o escopo delimitado (seleção de casos de teste). Os trabalhos sem avaliação de resultados e trabalhos cujo foco seja a geração, minimização ou priorização de casos de teste foram excluídos.

Para avaliar criticamente as evidências acerca do tópico em questão foram propostas as seguintes questões de pesquisa:

- Quais são os métodos empregados para seleção de casos de testes nos mais diferentes domínios de *software*?
- Como os métodos de seleção de casos de testes são avaliados?

Com o propósito de obter o panorama de pesquisa atual foram utilizadas as principais fontes disponíveis na *web*, identificadas em uma prévia análise exploratória. Tais fontes são mundialmente reconhecidas pela qualidade na produção literária e cobrem as principais revistas e eventos científicos na área de Computação, a saber:

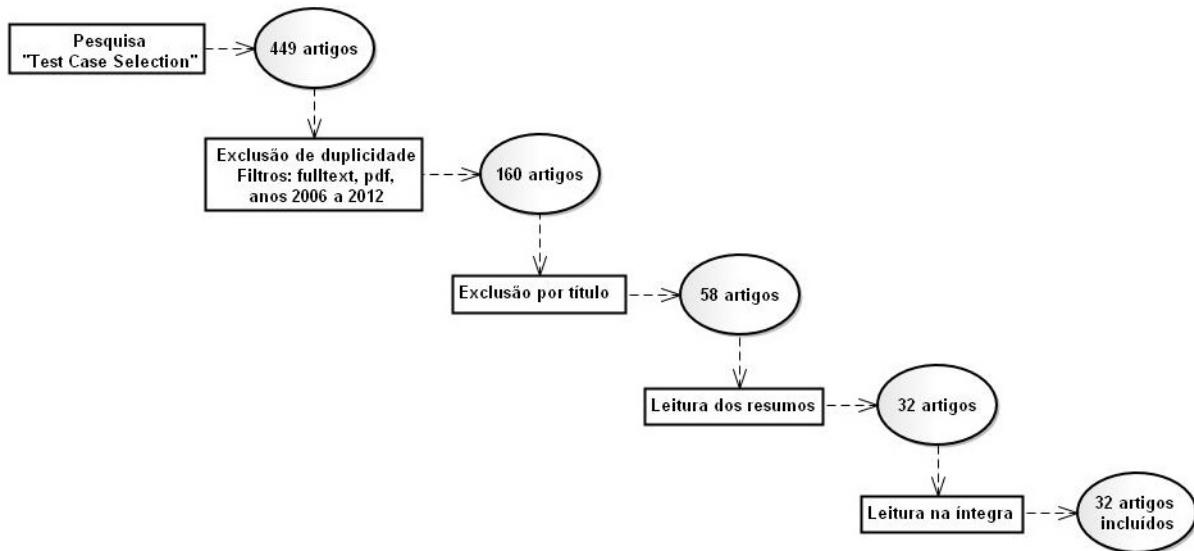
- Biblioteca Digital ACM (<http://portal.acm.org/>);
- Biblioteca Digital IEEE (<http://ieeexplore.ieee.org/Xplore/>);
- Biblioteca Digital Scopus (<http://www.scopus.com/home.url>);
- Biblioteca Digital Citeseer ([citeseer.ist.psu.edu](http://citeseer.ist.psu.edu)).

## 3.2 Condução da revisão sistemática

Para a pesquisa nas bases de dados científicas, utilizou-se a seguinte *string*: “*Test Case Selection*”. As máquinas de busca retornaram 449 artigos, todos escritos em inglês. O texto completo deveria estar disponível na *web* e a abrangência da pesquisa foi definida entre os anos 2006 e 2012 (inclusive). A Figura 14 descreve o procedimento de pesquisa. O retângulo representa uma ação, enquanto a elipse indica o resultado obtido.

É importante observar que as bases eletrônicas Scopus e Citeseer também cobrem alguns resultados da ACM e IEEE. Após a exclusão das duplicatas e da aplicação de filtros durante a consulta, restaram 160 artigos para análise. O processo de exclusão por títulos resultou em 58 artigos passíveis de inclusão e, após a leitura dos resumos e da leitura na íntegra, cada artigo foi selecionado se confirmada sua relevância. No final do processo

32 artigos atenderam aos critérios de inclusão preestabelecidos e foram definitivamente incluídos.



**Figura 14** – Processo da pesquisa em bases de dados científicas

A pesquisa no Citeseer retornou 9 artigos. Mesmo alterando o modo de procura, incluindo a string “*Test Case Selection*” e utilizando os campos “*title*”, “*text*” e “*keywords*”, sua máquina de busca retornou apenas um resultado passível de inclusão. A Tabela 4 resume os resultados para cada uma das bases de dados consultadas.

**Tabela 4** – Quantidade de artigos obtidos nas bases de dados indexadas

Fonte	# Resultados	# Artigos incluídos
ACM	189	12
Scopus	169	9
IEEE	82	10
Citeseer	9	1
<b>Total</b>	<b>449</b>	<b>32</b>

### 3.3 Resultados e discussões

A maioria dos artigos analisados foi publicada em conferências da área Engenharia de Software. Contudo é interessante notar que as pesquisas sobre seleção de casos de teste não se limitam a esta área da Computação. Há artigos publicados em veículos das áreas de inteligência artificial, computação genética e evolucionária, engenharia, ciência e gestão. A Tabela 5 apresenta os principais veículos de publicação.

**Tabela 5 – Principais veículos de publicação**

Conferências, Workshops e Simpósios	ACM Conference on Automated Software Engineering	59%
	ACM Conference on Genetic and Evolutionary Computation	
	ACM Symposium on Applied Computing	
	ACM Symposium on Foundations of Software Engineering	
	ACM Symposium on Software Testing and Analysis	
	ACM Workshop on Random Testing	
	ACM Workshop on Software Quality Assurance	
	ACM-W Conference – Celebration on Women in Computing in India	
	ACM/IEEE Symposium on Empirical Software Engineering and Measurement	
	ACM/IEEE Workshop on Random Testing/Conference on Software Engineering	
	ACM/IEEE Conference on Software Engineering	
	IEEE Conference on Computer and Information Science	
	IEEE Conference on Software Engineering and Data Mining	
	IEEE Conference on Software Testing, Verification and Validation	
	IEEE Conference on Tools with Artificial Intelligence	
	IEEE Conference on Computer Software and Applications	
	IEEE Conference on Advances in Engineering, Science and Management	
	IEEE Symposium on High Assurance Systems Engineering	
Journals	IEEE Transactions Software Engineering Wiley Software Testing, Verification and Reliability Elsevier Journal of Systems and Software	19%
Notes	ACM SIGSOFT Software Engineering Notes Springer Lecture Notes in Computer Science	19%
Outros	Elsevier Information Sciences Database On-line	3%

É evidente a importância das conferências, *workshops* e simpósios para a publicação de artigos científicos na área da Computação. Ambos representam 59% de todas as publicações incluídas nesta revisão sistemática. Os artigos foram publicados por 80 pesquisadores distintos, e a maioria foi produzida por 3 ou mais pesquisadores (69%). Alguns trabalhos apresentam mais de um estudo empírico de seleção de casos de teste e alguns pesquisadores participam de mais de um artigo. A Tabela 6 apresenta a síntese dos pesquisadores com duas ou mais publicações e sua respectiva área de pesquisa. Destacam-se a Universidade de Swinburne na área de *Adaptive Random Testing*, a Universidade de Oslo na área de *Similarity Approach for Model-based Test Case Selection* e a King's College London na área de *Multi-objective Test Case Selection*.

**Tabela 6 – Pesquisadores e suas publicações**

Autores	# Public	Universidade/País	Área de pesquisa
Chen, T. Y.	7	Swinburne, Austrália	<i>Random and Adaptive Random Testing</i>
Kuo, F. C.	5	Wollongong and Swinburne, Austrália	
Mayer, J.	3	Ulm, Alemanha	
Liu, H.	3	Swinburne, Austrália	
Huang, D. H.	2	Swinburne, Austrália	
Arcuri, A.	4	Oslo, Noruega	<i>Similarity Approach for Model-based TCS</i>
Hemmati, H.	4	Oslo, Noruega	
Briand, L.	3	Oslo, Noruega	
Harman, M.	2	King's College London, Inglaterra	<i>Regression Test, Multi-objective TCS</i>
Yoo, S.	2	King's College London, Inglaterra	

Para avaliar a qualidade dos artigos analisou-se algumas evidências que indicam facilidade na reprodução e na continuação dos trabalhos reportados. Para tanto, verificou-se em cada artigo a existência dos seguintes tópicos: o artigo reporta o código ou pseudocódigo da aplicação? Os pesquisadores apresentam as limitações de sua aplicação? Os artigos contêm a descrição dos trabalhos relacionados e a perspectiva de trabalhos futuros?

Os resultados mostram que o nível de detalhe dos artigos analisados é bastante diversificado. Em 57% dos artigos os pesquisadores reportam o código ou o pseudocódigo da implementação; 43% apresentam as limitações de suas aplicações; 64% dos artigos contêm a descrição de trabalhos relacionados e 50% mostram a perspectiva de trabalhos futuros. A inclusão de tais tópicos deve ser encorajada na escrita de artigos científicos voltados à área da Computação, pois fornecem uma visão mais abrangente e facilitam a análise e a reprodução dos métodos de seleção.

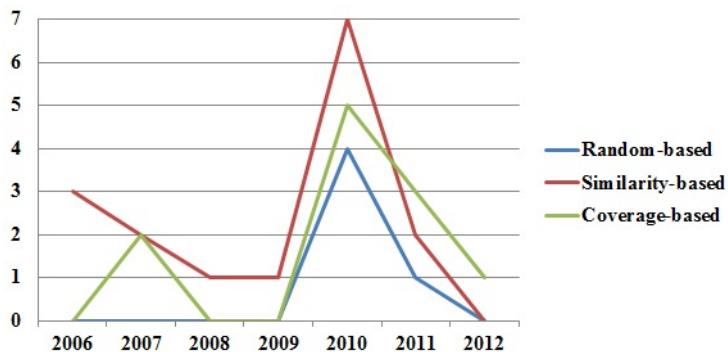
### 3.3.1 Visão geral das heurísticas de seleção de casos de teste

Um ponto importante a ser discutido nesta revisão sistemática é o fato de que grande parte dos métodos de seleção encontrados são baseados em heurísticas. As heurísticas podem ser entendidas como um processo no qual se adota uma hipótese prévia, como idéia diretriz, para a resolução de problemas [51].

As heurísticas não são determinísticas e se fundamentam na experiência ou julgamento do ser humano, portanto não há garantias de que a adoção de tais hipóteses sempre resulte em testes eficazes [51]. É possível, por exemplo, que um caso de teste dissimilar ou um caso de teste com maior cobertura não encontre qualquer falha.

As atividades de teste normalmente são complexas e envolvem múltiplos objetivos, tornando os métodos determinísticos muito difíceis de serem aplicados neste domínio. Em muitos casos é impossível, por exemplo, determinar previamente o resultado de um teste ou o esforço computacional requerido, fatos que tornam as abordagens heurísticas fundamentais nas pesquisas de seleção de casos de teste [52].

As três heurísticas mais reportadas nos últimos anos são a *random selection*, a *coverage-based selection* e a *similarity-based selection*, presentes em 16%, 31% e 47% dos artigos avaliados, respectivamente. A Figura 15 apresenta o comportamento das publicações em relação às heurísticas. Em 2010 houve um aumento considerável de publicações que abrangem esses tópicos.



**Figura 15** – As heurísticas de seleção de casos de teste mais abordadas

As subseções seguintes apresentam detalhes sobre os fundamentos das principais heurísticas de seleção de casos de teste encontradas nos artigos avaliados.

### 3.3.1.1 Heurística de seleção aleatória

A heurística de seleção aleatória (*random selection*) segue a hipótese de que a seleção aleatória com base em probabilidades é capaz de prover um teste eficaz, além de ser um método de simples implementação e que requer um baixo custo computacional para sua execução. Na maioria dos casos, a seleção aleatória é realizada com probabilidade uniforme dentro do domínio de teste. Entretanto, alguns métodos de seleção aleatória objetivam encontrar um grupo de dados para testes capazes de representar a expectativa de uso do programa, ou seja, o perfil operacional [22]. Os perfis operacionais permitem a definição de um espaço amostral de dados para teste condizente com os requisitos a serem testados e, neste contexto, é possível associar probabilidades à seleção dos casos de teste [22].

### 3.3.1.2 Heurística de seleção utilizando conceitos de similaridade

O principal fundamento da seleção de casos de teste utilizando conceitos de similaridade é a hipótese de que a diversificação dos casos de teste tende a maximizar a capacidade de detecção de falhas.

Essa abordagem envolve a utilização de funções de similaridade para mensurar a diferença entre os casos de teste. Uma vez definida a função de similaridade, o método de seleção escolhe um subconjunto reduzido e dissimilar de pares de casos de teste, otimizando o conjunto de testes.

### 3.3.1.3 Heurística de seleção utilizando critérios de cobertura

A seleção de casos de teste que utiliza critérios de cobertura tem o objetivo de maximizar a cobertura do teste, normalmente em termos de cobertura de modelo ou de código. Essa heurística segue a hipótese de que os casos de teste com maior cobertura tendem a ser mais eficazes na detecção de falhas [42]. Infelizmente, não há garantias de que haja alguma ligação direta entre a cobertura e a capacidade de detecção de falhas antes da execução dos testes; portanto, é natural que se complemente a cobertura com critérios adicionais [53].

Nesse sentido, Mirarab et al. [54] apresentam uma aplicação baseada em um método multicritério iterativo que foca partes isoladas do *software* e utiliza várias técnicas complementares para maximizar a cobertura do código. Yoo e Harman [53] e Souza et al. [55] apresentam métodos multicritério que consideram tanto a cobertura (de requisitos ou código) quanto o custo da execução para selecionar os casos de teste. Outras abordagens são discutidas por Tsai et al. [56], que apresentam uma proposta de seleção a partir do modelo chamado *Coverage-Relationship Model*, capaz de selecionar e ordenar os casos de teste, bem como eliminar os dados que possuem a mesma capacidade de cobertura do modelo.

### 3.3.2 Visão geral dos métodos de seleção de casos de teste

Esta seção apresenta os resultados que permitem responder a primeira questão de pesquisa: quais são os métodos empregados para seleção de casos de testes nos mais diferentes domínios de *software*? Algumas peculiaridades dos métodos mais reportados também são discutidas adiante.

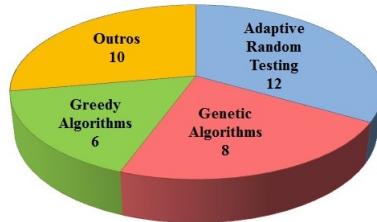
A Tabela 7 apresenta uma visão global dos métodos de seleção, na qual cada artigo é referenciado de acordo com os métodos reportados. Muitos métodos de seleção utilizam heurísticas, discutidas na seção 3.3.1, as quais são apresentadas na terceira coluna da Tabela 7, quando for o caso.

Dentre os 32 artigos analisados, foram identificados 18 métodos distintos, apresentados na primeira coluna da Tabela 7. Essa elevada proporção de métodos diferenciados (18 métodos em 32 artigos) indica que os pesquisadores tendem a propor novos estudos de seleção de casos de teste ao invés de continuar o trabalho de outros pesquisadores. Esse resultado é semelhante às conclusões apresentadas por Engstrom et al. [76], porém não se restringe a métodos de seleção utilizados em testes de regressão.

**Tabela 7** – Visão global dos métodos de seleção de casos de teste

Métodos de seleção de casos de teste	Artigos	Heurística
Random Testing	[22], [35], [42], [44], [55]	Random Selection
Adaptive Random Testing	[22], [44], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67]	Similarity-based Selection
Genetic Algorithms	[35], [44], [62]	
Greedy Algorithms	[35], [42]	
Agglomerative Hierarchical Clustering	[6], [44]	
Similarity-based Selection for Model-based Testing	[21]	
Genetic Algorithms	[22], [35], [42], [57]	Coverage-based Selection
Greedy Algorithms	[35], [42], [44]	
Coverage Relationship Model	[56]	
Particle Swarm Optimization	[55]	
Greedy Algorithms e Pareto Efficient Genetic Algorithms	[53]	
Pareto Efficient Genetic Algorithms	[68]	
Integer Programming e Greedy Algorithms	[54], [74]	
Seleção de casos de testes a partir de Modelos	[69]	Não há
Métodos Estatísticos: Chi-Quadrado e P-Measure	[70]	
Lógica Fuzzy	[71]	
Component Dependency Model (CDM)	[72]	
Orientado a eventos. Utiliza grafo evento/dependência	[73]	
Surveys, revisões sistemáticas ou pesquisas qualitativas	[4], [38], [75], [76]	

A Figura 16 apresenta um gráfico com os métodos mais reportados em estudos de seleção de casos de teste.

**Figura 16** – Métodos mais utilizados em seleção de casos de teste

Vários pesquisadores abordam mais de um método de seleção para a realização de seus experimentos, principalmente para comparação e validação das propostas. Por exemplo, Hemmati e Briand [42] utilizam os métodos *adaptive random testing* (ART) e *genetic algorithm* (GA) para avaliar as funções de similaridade e os compararam com os métodos *greedy-search coverage* em termos de capacidade de detecção de falhas. Já Souza et al. [55] propõem um novo método com base em *particle swarm optimization* e o compara com o *random selection*. As próximas subseções apresentam uma análise detalhada sobre os métodos de seleção mais reportados, dispostos Tabela 7.

### 3.3.2.1 Método *adaptive random testing* (ART)

O método *adaptive random testing* (ART) surgiu como uma extensão otimizada do *random testing*. A teoria por trás do ART é fundamentada em estudos que tratam sobre o

comportamento de certos tipos de falhas em *software* de entrada numérica. Tais estudos empíricos comprovam que distribuir os casos de teste de maneira uniforme por todo o domínio de entrada tende a aumentar a eficácia em termos de detecção de falhas [75].

A premissa básica do ART é que se a execução de um grupo “A” de casos de testes produz um resultado de saída correto, casos de teste alocados em regiões distantes do grupo “A” têm maior probabilidade de revelar falhas [75]. A explicação para este fenômeno pode ser encontrada em vários experimentos que corroboram as seguintes conclusões [22, 44, 58–61, 63–67]: 1) regiões do código-fonte que apresentam erros geralmente estão relacionadas a regiões contíguas do domínio de entrada capazes de revelar falhas e 2) regiões do código-fonte que não apresentam erros geralmente estão relacionadas a regiões contíguas do domínio de entrada que produzem um resultado de saída correto.

Em suma, os casos de teste similares tendem a revelar os mesmos padrões de falhas. Baseado neste pressuposto, o ART utiliza algum parâmetro capaz de computar a diferença entre os casos de teste para garantir uma melhor distribuição pelo domínio de entrada. Inicialmente, os casos de teste são escolhidos de maneira aleatória a partir do conjunto de testes original e então o método seleciona apenas aqueles que atendam ao critério de distância preestabelecido. O processo se repete até atingir o ponto de parada desejado. Um possível pseudocódigo para o ART é apresentado no Algoritmo 2 [44]:

```

Entrada: Z ← ∅ (representa o conjunto de teste)
1 Gerar aleatoriamente um caso de teste t e adicionar em Z
2 Testar o programa utilizando t
3 enquanto ponto de parada não atingido faca
4   Distância D = 0
5   Gerar aleatoriamente casos de teste  $c_1 \dots c_i$  até atingir o tamanho da amostra
6   para cada caso de teste  $c_i$  hacer
7     Calcular a distância mínima d em relação a Z
8     se  $d > D$  então
9       D = d
10      Adicionar  $c_i$  em Z
11      Testar o programa utilizando  $c_i$ 
12 Fim da execução
```

**Algoritmo 2:** Adaptive random testing

O ART tem se mostrado um método eficaz para teste em programas de domínio de entrada numérico, cujo ponto-chave é utilizar algum critério de interesse para distribuir os dados de teste de maneira uniforme pelo domínio de entrada [61, 67]. Contudo, em situações reais, o domínio de entrada/saída do *software* pode se estender aos mais vari-

ados tipos de dados. Nestes casos, o principal desafio é definir medidas alternativas que permitam a distribuição uniforme dos casos de teste [75]. Para tanto, o método ART utiliza conceitos de similaridade visando a garantir a diversidade entre os casos de teste. O princípio é simples: presumindo que os casos de teste similares tendem a revelar os mesmos padrões de falhas, as funções de similaridade devem calcular a distância entre eles e selecionar para execução apenas os casos de teste mais dissimilares.

### 3.3.2.2 Método *similarity-based selection*

No método ART, os casos de teste são escolhidos aleatoriamente a partir do conjunto de testes original e posteriormente se realiza uma avaliação dos resultados, de forma que apenas os casos de teste que atendam ao critério de dissimilaridade sejam aproveitados.

De fato, esse é um método de seleção empregado em testes funcionais (caixa-preta), pois a única informação utilizada sobre o SUT é a premissa de que regiões do código-fonte que apresentam erros geralmente estão relacionadas a regiões contíguas do domínio de entrada capazes de revelar falhas. Por outro lado, quando existem informações adicionais sobre o *software* sob teste, é natural utilizá-las visando agregar valor ao teste.

Nota-se que, ao tratar de conceitos de similaridade com o objetivo de garantir a diversidade entre os casos de teste, o que se pretende é encontrar mecanismos capazes de mensurar a distância entre os pares de casos de teste. Para se calcular essas distâncias é usual a implementação de funções de similaridade apropriadas para representação e manipulação dos dados de entrada [44]. Tais funções devem ser capazes de mensurar a similaridade dos casos de teste, bem como garantir a eficiência na execução do método. Em muitos casos, a mensuração da similaridade é aplicada em termos de cobertura de código [44], cobertura de modelo [62] ou distância entre dados numéricos e alfanuméricos [75].

Hemmati e Briand [42] realizaram um estudo empírico sobre a eficácia de várias funções de similaridade utilizadas em métodos de seleção de casos de teste com base em modelos, tais como *Jaccard Index*, distância de *Hamming*, função *Counting*, *Needleman-Wunsch* e distância de *Levenshtein*. No processo de teste com base em modelos, o domínio dos dados de entrada é definido por modelos – como o diagrama de máquinas de estado da *Unified Modeling Language* (UML) – e representado por seus elementos (e.g. *triggers*, *round trip paths*, *transitions*, outros). Os modelos são uma descrição abstrata e representam a estrutura e/ou funcionalidades do *software*. Tais descrições permitem a geração de casos de teste abstratos e, por meio de métodos de seleção, apenas aqueles que atendam

aos critérios de interesse são instanciados e executados.

Em um teste com base em máquinas de estado, por exemplo, um caso de teste abstrato pode ser representado por um caminho (*path*) da máquina de estado e instanciado com diferentes parâmetros de entrada [62]. Existem situações nas quais os métodos para geração de casos de teste com base em modelos utilizam critérios de cobertura estrutural ou funcional aplicados de forma exaustiva, gerando uma grande quantidade de casos de teste [21]. Nessas situações, métodos de seleção com base em similaridade podem ser utilizados para otimização do conjunto de testes.

### 3.3.2.3 Métodos *search-based*

A seleção de casos de teste pode ser tratada como um problema de otimização nas quais os métodos *search-based* exploram um espaço de possíveis soluções objetivando encontrar as melhores combinações de resultados com um custo computacional aceitável [44]. Os algoritmos genéticos e o *greedy algorithm*, subclasses dos métodos *search-based*, são os mais empregados nesse domínio.

O algoritmo genético é um tipo de algoritmo para descoberta de aproximações progressivas e é definido por quatro principais elementos: população, seleção, *crossover* e mutação. O Algoritmo 3 define a estrutura básica do algoritmo genético [43].

<b>Entrada:</b> População $P(t)$ de tamanho $n$ <b>1 enquanto</b> permanecer condição <b>faça</b> <b>2</b> Selecionar subpopulação $p(t)$ de $P(t)$ de acordo com o critério de seleção <b>3</b> $p'(t) \leftarrow crossover p(t)$ de acordo com o critério de <i>crossover</i> <b>4</b> Realizar mutação em $p'(t)$ de acordo com o critério de mutação <b>5</b> Avaliar $p'(t)$ em relação à $p(t)$ de acordo com o critério de avaliação <b>6</b> <b>se</b> $p'(t)$ for melhor que $p(t)$ <b>então</b> <b>7</b> $p(t) \leftarrow p'(t)$ <b>8</b> Fim da execução
---

**Algoritmo 3:** Genético

A execução do algoritmo genético pode ser descrita da seguinte maneira:

1. A população são as possíveis soluções (ou domínio), representadas por indivíduos. Os indivíduos também são conhecidos como cromossomos.
2. A cada iteração (geração) é realizada a seleção criteriosa de um subconjunto de soluções da população.

3. A subpopulação é utilizada pelo operador *crossover* para combinar parte dos cromossomos utilizando como critério probabilidades pré-definidas, produzindo novas soluções chamadas “*offspring*”.
4. O operador de mutação realiza algumas mudanças nos cromossomos do *offspring* para otimização dos resultados.
5. A função de *fitness* avalia as novas soluções *offspring*. Se o resultado for satisfatório, tais soluções tornar-se-ão parte da população na próxima iteração.

O ponto-chave do algoritmo genético é a definição da função de *fitness*. Em aplicações de seleção de casos de teste, normalmente os indivíduos são expressos por um conjunto ou uma sequência de dados de tamanho  $n$ , em que a função de *fitness* deve ser ajustada aos critérios de teste com o objetivo de avaliar o quanto bem o conjunto de dados se aproxima das especificações [52].

Os algoritmos genéticos são muito utilizados em situações em que o domínio de entrada é definido por modelos. Nesse caso, o algoritmo visa a selecionar a melhor combinação de casos de teste abstratos (em termos de dissimilaridade) para posterior instância/execução [35, 44, 62]. Ele também é eficaz para a otimização utilizando diferentes heurísticas e critérios de seleção, tais como *pareto-optimalidade* [53], *mutation-score* e *path-coverage* [57].

O *greedy algorithm* é um algoritmo que segue a heurística de fazer a escolha ótima local em cada repetição, com a expectativa de possivelmente encontrar um ótimo global. Esta abordagem começa com um subconjunto vazio do conjunto de teste e iterativamente acrescenta um caso de teste mais vantajoso (que proporciona maior cobertura do que os restantes, por exemplo). O processo se repete até que o ponto de parada desejado seja alcançado [68]. O pseudocódigo do *greedy algorithm* é dado pelo Algoritmo 4 [77]:

1	<b>Entrada:</b> C representa o conjunto de testes
2	<b>enquanto</b> Houver casos de teste em C e não houver solução (S) <b>faça</b>
3	x ← selecionar o melhor elemento em (C)
4	<b>se</b> não for possível encontrar o melhor elemento <b>então</b>
5	<b>retorna</b> “não há solução”
6	<b>senão</b> S ← S ∪ {x}
7	<b>retorna</b> S
8	Fim da execução

**Algoritmo 4:** Greedy

Apesar da popularidade do algoritmo genético e do *greedy algorithm*, outros métodos *search-based* vêm ganhando espaço na literatura voltada à Engenharia de Software, tal como o método *particle swarm optimization* [55].

### 3.3.2.4 Métodos de *clustering*

Os métodos de *clustering* empregam critérios para identificar, classificar e agrupar os casos de teste de acordo com as suas características, minimizando a variância entre os elementos do mesmo grupo e maximizando a variância entre os elementos de grupos distintos [44]. O benefício da utilização de *clustering* reside no fato de que a classificação e a obtenção de grupos condensados de dados permite aumentar o poder de inferência sobre os conjuntos observados. Independentemente do algoritmo utilizado, a maioria dos métodos de *clustering* adota medidas de aproximação ou de distância para definir os *clusters*. Sapna e Mohanty [6] e Hemmati et al. [44] utilizaram o *Agglomerative Hierarchical Clustering* (AHC) e funções de similaridade para selecionar os casos de teste mais dissimilares. Nessa abordagem, uma vez definidos os *clusters*, utilizam-se métodos adicionais (como o método *random selection*, por exemplo) para selecionar os melhores casos de teste de cada *cluster*.

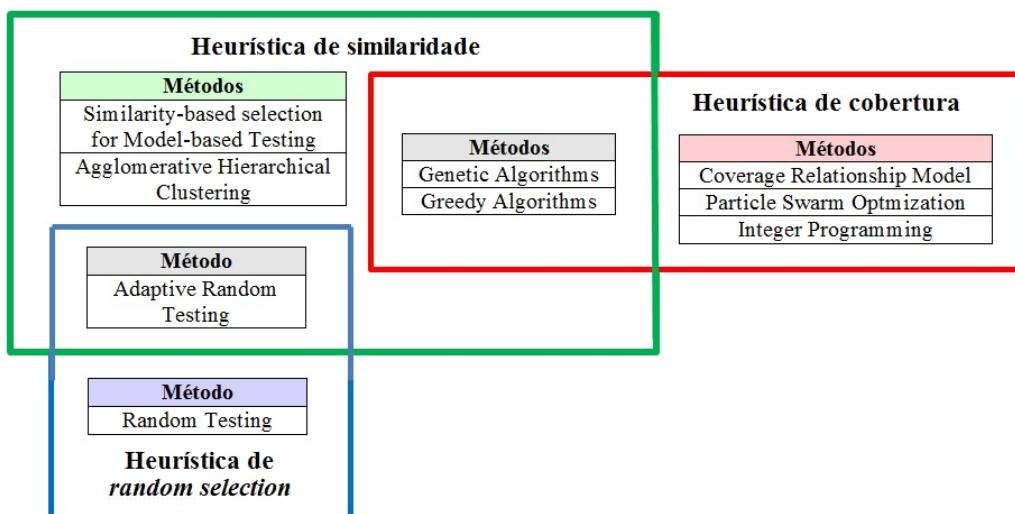
### 3.3.2.5 Métodos estatísticos e lógica *Fuzzy*

A seleção de casos de teste pode ser tratada como um problema no qual os dados de teste devem ser escolhidos com base na probabilidade do acontecimento de algum evento de interesse (e.g. probabilidade de detectar falhas). Em Yu et al. [70], a abordagem para testes de regressão envolve a caracterização, classificação e seleção dos casos de teste utilizando testes estatísticos por meio de Qui-quadrado e um modelo de *Support Vector Machine* (SVM).

Outra abordagem possível consiste em emular algumas características geralmente não determinísticas e imprecisas do ser humano, como os processos de julgamento e as tomadas de decisões. Em Xu et al. [71], a abordagem utiliza lógica *Fuzzy* e considera a correlação de vários aspectos para a seleção, tais como o perfil do cliente, análise de execução dos casos de teste, taxa de falhas e mudanças na arquitetura do sistema.

### 3.3.3 Relação entre métodos e heurísticas de seleção de casos de teste

A Figura 17 representa a relação entre as heurísticas e os métodos de seleção dispostos na Tabela 7. Os retângulos representam os grupos de heurísticas e suas tabelas internas descrevem os métodos de seleção empregados. Esse relacionamento é importante, pois alguns métodos de seleção (e.g. *greedy algorithm*) podem ser implementados utilizando diferentes heurísticas. Por exemplo, Hemmati e Briand [42] aplicam o algoritmo genético em testes baseados na heurística de similaridade e de cobertura concomitantemente.



**Figura 17** – Relação entre as heurísticas e os métodos de seleção de casos de teste

### 3.3.4 Avaliação dos métodos de seleção de casos de teste

A segunda questão de pesquisa consiste em descobrir como os métodos de seleção de casos de testes são avaliados. Dos 32 artigos incluídos, 28 apresentam experimentos com avaliações quantitativas de resultados. Existem 71 experimentos controlados (estudos de caso) utilizados como forma de avaliação dos resultados em 72% dos artigos analisados. Foram identificados testes em 65 *software*, classificados de acordo com o seu tipo e tamanho e dispostos conforme apresentados originalmente pelos pesquisadores. Dentre os experimentos, 40 *software* foram desenvolvidos pelos próprios pesquisadores (classificados como experimentais) ou obtidos em bases de dados na *web* – como a *Free Software Foundation* [39, 79] – e 25 *software* são reportados como aplicações industriais (e.g. Siemens e Space [21, 53]), dentre os quais alguns estão disponíveis no *Software Infrastructure Repository* (SIR) [80]. A Tabela 8 apresenta a visão global das avaliações dos métodos de

seleção de casos de teste.

**Tabela 8** – Tipos de programas utilizados em avaliações dos métodos de seleção

Estudo	Tipo de programa	Tamanho	Quantidade
Estudo de Caso	Experimental	Pequeno	38
		Grande	2
	Industrial	Pequeno	21
		Grande	4
		<b>Total</b>	<b>65</b>

Grande parte dos experimentos foi conduzida por meio de testes em *software* considerados pequenos. Dentre os 65, apenas seis experimentos foram realizados em *software* de larga escala, dos quais quatro são voltados a aplicações industriais. A maioria dos experimentos utilizam *software* escritos em Java, C ou C++, encontrados em [22, 39, 44, 54, 57, 59, 60, 65, 66, 71], *software* embarcados [44, 69], *mobile phone* [21], *web services* [56] e outras aplicações *web* [73]. Alguns métodos de seleção requerem tipos específicos de entradas, como a OCL (*Object Constraint Language*) [22].

Em relação à avaliação dos experimentos, os pesquisadores utilizam diferentes métricas para determinar a eficácia e a eficiência dos métodos empregados, como a quantidade de casos de teste requeridos para encontrar a primeira falha, a taxa de detecção de falhas e a cobertura de código ou de modelo. A Tabela 9 apresenta as principais métricas para avaliação dos métodos de seleção de casos de teste.

**Tabela 9** – Principais métricas para avaliação de métodos de seleção de casos de teste

Métrica	Artigos	% Artigos
# CT requeridos para encontrar a primeira falha <sup>†</sup>	[22], [58], [59], [60], [61], [63], [64], [65], [66], [67]	36%
Taxa de detecção de falhas	[6], [21], [44], [57], [70], [71]	21%
Cobertura de código ou de modelo	[21], [53], [54], [55], [57]	18%
Tempo ou custo de execução	[53], [55], [70]	11%

Muitos pesquisadores adotam a quantidade de casos de testes requeridos para encontrar a primeira falha como a principal métrica de avaliação dos métodos de seleção. A ideia geral é que uma falha em um determinado ponto do código pode causar outras falhas em diferentes pontos do código. Portanto, ao encontrar a primeira falha, o desenvolvedor deve reparar o defeito causador da falha antes de submeter o *software* a um novo teste,

<sup>†</sup>Incluindo *F-measure*, que é a quantidade média de casos de testes requeridos para encontrar a primeira falha em testes aleatórios.

evitando outras falhas decorrentes da primeira [78]. Em contraste, a segunda métrica mais utilizada é a taxa de detecção de falhas, que considera todas as falhas encontradas.

### 3.4 Trabalhos relacionados

Engstrom et al. [76] elaboraram uma revisão sistemática sobre métodos de seleção de casos de teste para testes de regressão. Os pesquisadores selecionaram 28 dentre 2923 artigos encontrados em diversas fontes de dados científicas. Com base na análise realizada, eles concluíram que as pesquisas disponíveis não são suficientemente avaliadas para que os desenvolvedores possam utilizá-las para tomada de decisões. Em muitos estudos, apenas um dos problemas é avaliado e o contexto é muito específico para que seja aplicado diretamente pelos desenvolvedores. Os pesquisadores também não encontraram evidências que permitam determinar a superioridade de algum método específico em relação a todos os métodos avaliados.

Yoo e Harman [38] escreveram um *survey* sobre as abordagens de minimização, priorização e seleção de casos de teste em testes de regressão. Os pesquisadores discutem diversos métodos utilizados para reduzir o conjunto de testes, a interrelação entre as abordagens, as características, as vantagens e as desvantagens de cada tópico em questão.

Chen et al. [75] apresentaram uma síntese de importantes estudos teóricos relacionados ao *adaptive random testing*. Os pesquisadores discutem tópicos relevantes como padrões de falhas, limites teóricos, sequências aleatórias e testes com base em falhas.

Um *survey* sobre como as organizações adotam a seleção de casos de teste em seus projetos de desenvolvimento de *software* foi apresentado por Kasurinen et al. [4]. Os pesquisadores classificaram as estratégias das organizações em dois grupos: 1) *risk-based*, cujo foco é testar as partes do *software* que são consideradas muito caras para reparar depois da distribuição final e 2) *design-based*, na qual o objetivo dos testes visa a garantir que o *software* é capaz de realizar as tarefas para o qual foi projetado, em termos de funcionalidades. O estudo incluiu entrevistas com 36 profissionais de 12 organizações de desenvolvimento de *software*, visando a descobrir alguns fatores que influenciam na escolha da estratégia de seleção. Os resultados apontam que *risk-based* é a estratégia economicamente viável em casos nos quais é preferível manter os prazos ao invés de prolongar a fase de testes. A estratégia *design-based* é mais utilizada em organizações que dispõem de amplos recursos para testes.

A diversidade de métodos reportados e a difusão nas mais variadas áreas da Com-

putação mostram que a seleção de casos de teste tem sido muito pesquisada, analisada e avaliada a ponto de existirem soluções maduras, viáveis e bem estabelecidas. No entanto, não foram encontrados estudos que apresentem métodos de seleção de casos de teste para *software* com entradas/saídas complexas, como os *software* de processamento de imagens.

### 3.5 Tendências e desafios

A seleção de casos de teste, como parte integrante do processo de desenvolvimento de *software*, normalmente envolve diferentes critérios e objetivos de teste. Devido à complexidade de aplicações do mundo real, considerar tão somente a capacidade de redução do conjunto de testes ou a capacidade de detecção de falhas pode, em muitos casos, não ser suficiente para garantir a viabilidade de um método. Um único caso de teste que seja capaz de disparar muitos processos pode levar vários minutos para ser executado e, nesse cenário, o custo computacional e o tempo de execução podem ser cruciais para determinar a aplicabilidade e a eficiência do método de seleção.

Os resultados mostram que a maioria dos trabalhos reportados apresenta uma única métrica (e.g. cobertura de código ou *F-measure*) como forma de avaliação dos experimentos. Os pesquisadores consideram mais de uma métrica concomitantemente em apenas cinco artigos, nos quais Cartaxo et al. [21] e Mala e Mohan [57] avaliam a capacidade de detecção de falhas e a cobertura de código e Yu et al. [70] consideram a capacidade de detecção de falhas e o tempo de execução dos casos de teste. Yoo e Harman [53] e Souza et al. [55] apresentam métodos que consideram tanto a cobertura (de requisitos ou código) quanto o custo da execução para selecionar os casos de teste. A pesquisa de métodos multicritérios para a seleção de casos de teste deve ser encorajada, pois o seu emprego pode contribuir para a otimização da escalabilidade, eficiência, robustez e confiabilidade dos testes.

Grande parte dos experimentos avaliados foi realizada em *software* desenvolvidos pelos próprios pesquisadores e classificados como pequenos. A análise e a validação de resultados em experimentos de seleção de casos de teste ainda continuam limitadas. Apesar da existência de fontes importantes como o *Free Software Foundation* [79] e o SIR [80], há a necessidade de expandir os experimentos em termos de tamanho, aplicação (e.g. industriais) e complexidade do *software*, pois replicar os estudos utilizando tais alternativas pode revelar diferentes padrões de efetividade e eficiência, bem como ajudar a comprovar a viabilidade e a aplicabilidade dos métodos de seleção propostos.

A maioria dos métodos de seleção analisados foca um contexto ou domínio específico de aplicação. Devido a essa característica, não foram encontrados indícios que permitam avaliar a superioridade de qualquer método sobre outro. Há poucos trabalhos sobre métodos de seleção focados em *web services* ou *mobile phone* e não foram encontradas pesquisas voltadas a *software* que utilizam *Graphical User Interface* (GUI) ou *software* de domínios complexos (e.g. processamento de imagens). Nesse cenário, garantir a escalabilidade, expandir a capacidade de generalização e desenvolver *frameworks* ou ferramentas de suporte à seleção de casos de teste ainda são grandes desafios a serem vencidos pela comunidade científica.

## 3.6 Limitações da pesquisa

Durante a fase inicial de pesquisa desta revisão sistemática surgiram algumas questões óbvias quanto ao escopo do trabalho, mas que precisam ser esclarecidas: 1) quais as bases de dados científicas mais adequadas para a realização da pesquisa? 2) como determinar um período de publicação suficientemente relevante para o trabalho? 3) quais as *strings* mais apropriadas para pesquisa nas bases de dados científicas?

As duas primeiras questões foram decididas por meio de uma prévia análise exploratória em diversos veículos de publicação científica. A terceira questão leva a uma discussão controversa e nada trivial: a padronização dos termos utilizados em trabalhos voltados à área de Computação. O termo “seleção de casos de teste” e sua distinção de outras terminologias voltadas à redução do conjunto de testes não é absolutamente consensual entre os pesquisadores. Nesse cenário, é natural que existam outras terminologias para descrever métodos similares aos apresentados nessa RS. Tendo em vista que o objetivo desse trabalho inclui a pesquisa e a análise de artigos que apresentem métodos de redução do conjunto de testes voltados aos testes de regressão e métodos cujo objetivo é a seleção de casos de teste com base em heurísticas, houve a necessidade de excluir outros estudos relevantes cujo foco não abrangia o escopo da pesquisa.

Enfim, os três fatores apresentados foram considerados e representam uma limitação à validação da pesquisa. Por outro lado, o escopo do trabalho é suficientemente abrangente e apresenta uma perspectiva relevante sobre o estado da arte de seleção de casos de teste.

## 3.7 Conclusões da revisão sistemática

A pesquisa apresentada nesse capítulo focou a obtenção do estado da arte do domínio, além de possibilitar a organização de ideias, a summarização de informações relevantes e a auditoria.

Foram identificados 18 métodos distintos em 32 artigos analisados e, por meio de uma avaliação empírica, constatou-se que: 1) os métodos mais reportados são o *adaptive random testing*, os algoritmos genéticos e o *greedy algorithm* 2) a maioria dos métodos de seleção utiliza heurísticas como a diversidade em casos de teste e a cobertura de código ou de modelo 3) a maioria dos experimentos reportados utilizaram testes em pequenos *software*, alguns disponíveis no *Free Software Foundation* e no *Software Infrastructure Repository* e 4) a métrica mais utilizada para avaliar os experimentos de seleção é a quantidade de casos de teste necessários para encontrar a primeira falha; em contraste, a segunda métrica mais utilizada é a taxa de detecção de falhas. As evidências também indicam que os pesquisadores tendem a realizar estudos empíricos com base em novas abordagens ao invés de continuar o trabalho de outros pesquisadores.

Por fim, a maioria dos métodos de seleção avaliados foca um contexto ou domínio específico de *software*, por isso não foram encontradas evidências que permitam avaliar a superioridade de algum método em relação aos demais. Alguns desafios que ainda não foram superados e que, por isso, podem constituir importantes objetos de pesquisa são 1) avaliar os métodos e as métricas de seleção pesquisados em diferentes contextos para comprovar a sua efetividade, eficiência, aplicabilidade e escalabilidade; 2) desenvolver metodologias de seleção que possam ser estendidas para os mais diversos domínios de *software* e 3) desenvolver *frameworks* ou ferramentas de suporte à seleção de casos de teste.

## 3.8 Considerações finais

Nesse capítulo foi apresentada uma pesquisa acerca dos métodos para seleção de casos de testes, com o objetivo de obter o estado da arte neste domínio. O trabalho focou os temas atuais de métodos de seleção ao incluir artigos publicados entre 2006 e 2012 (sete anos).

É importante ratificar que não foram encontradas pesquisas que envolvam a utilização de CBIR para seleção de casos de teste, nem pesquisas que tratem sobre métodos de seleção para testes para *software* de processamento de imagens. No entanto, é possível verificar que existem métodos de seleção que utilizam os mesmos conceitos de similaridade encontrados em alguns sistemas de CBIR. Por exemplo, várias funções de similaridade avaliadas por Gonçalves et al. [19] para encontrar as imagens mais parecidas são as mesmas utilizadas no trabalho de Hemmati e Briand [42], Kuo et al. [61] e Mayer [64] para a seleção de casos de teste (e.g. distância Euclidiana, distância *Manhattan*, outras).

Por fim, essa revisão sistemática permite avaliar, empiricamente, que a utilização das funções de similaridade e dos métodos de *clustering* têm grande potencial de aplicação no que tange à seleção de casos de teste para *software* de processamento de imagens, nos quais os casos de teste são as próprias imagens. Tal conclusão tem como base o fato de que tais métodos são comprovadamente eficazes para a seleção de casos de teste em *software* sob outros domínios de aplicação (e.g. paradigma de OO, *software* embarcado, entre outros).

# *Capítulo 4*

## *Seleção de Casos de Teste Utilizando Conceitos de CBIR*

As metodologias utilizadas durante a fase de testes de *software* têm impacto direto na qualidade final do produto. Os testes manuais e sem planejamento podem implicar na produção de *software* de confiabilidade duvidosa e que não atendem aos requisitos almejados. Criar mecanismos automatizados que explorem os requisitos do *software* com o menor esforço possível é um grande desafio e, nesse cenário, as ferramentas de seleção de casos de teste são muito importantes para a definição das estratégias de teste.

Existem muitas abordagens de seleção de casos de teste focadas em *software* sob o paradigma de orientação a objetos, *software* embarcado e sistemas com entradas/saídas alfanuméricas em geral [38, 76]. No entanto, há uma lacuna quando se trata de seleção de casos de teste para *software* de domínios complexos, como os sistemas de processamento de imagens. Testar um sistema utilizando uma grande quantidade de imagens exige um alto custo computacional e, considerando a inviabilidade dessa alternativa, é apresentada uma nova abordagem para seleção de casos de teste intitulada *Tcs&CbIR*. A abordagem utiliza conceitos de *Content-Based Image Retrieval*(CBIR) para selecionar um subconjunto de casos de teste com alta probabilidade de revelar falhas, otimizando a execução dos testes.

Para alcançar o objetivo proposto, o presente capítulo foi organizado da seguinte maneira: a seção 4.1 apresenta a descrição da problemática de seleção de casos de teste no contexto do método *Tcs&CbIR*; a seção 4.2 mostra a metodologia e as definições gerais da abordagem proposta e a seção 4.3 conclui o capítulo.

### **4.1 Descrição do problema**

Uma possível maneira de identificar a presença de qualquer falha em um *software* sob teste é submetê-lo à execução usando todo o seu domínio de entrada, ou seja, submetê-lo a todas as possíveis condições válidas e inválidas utilizadas como parâmetros de entrada. Tendo em vista que a capacidade de detecção de falhas do domínio de entrada é plena,

torna-se razoável inferir que, idealmente, um conjunto de teste deveria contemplar todo o domínio de entrada. Entretanto, é comum que o domínio de entrada cresça conforme a evolução dos requisitos do *software*. Ainda é comum que seja inviável ou impossível obter casos de teste que cubram todo o domínio de entrada. Tal fenômeno é mais evidente quando o *software* em questão utiliza imagens como dados de entrada, pois o seu domínio é complexo.

Para elucidar tais assertivas, será utilizado como exemplo um *software* fictício desenvolvido para tratamento de imagens binárias (preto e branco) de tamanho de *10 linhas por 10 colunas (100 pixels)*, no padrão RGB. Considerando apenas as possibilidades válidas, cada *pixel* pode assumir dois valores distintos (0-0-0) ou (255-255-255). Portanto o domínio de entrada é composto por  $2^{100} \cong 1,267 \cdot 10^{30}$  imagens distintas. Agora, caso o *software* evolua para suportar imagens de tamanho de *100 linhas por 100 colunas (10000 pixels)*, o crescimento do domínio de entrada válido torna-se vertiginoso ( $2^{1000} \cong 1 \cdot 10^{301}$ ).

O exemplo citado mostra que, em linhas gerais, é inviável que o conjunto de teste disponível ao testador comporte todo o domínio de entrada de programas de processamento de imagens. Normalmente, o custo computacional para se testar todas as possibilidades de um *software* é conhecido e pode ser resolvido em tempo polinomial, mas sua execução é impraticável devido ao tamanho do domínio. Devido à essa característica, torna-se muito difícil, em termos práticos, o emprego de soluções determinísticas para otimizar a execução dos testes.

Uma alternativa para mitigar esse problema consiste em utilizar soluções de otimização computacional, geralmente não determinísticas, para selecionar os melhores elementos dentre um conjunto de teste  $T$ . Nesse cenário, a qualidade dos elementos pode ser mensurada pela função  $f(x)$  (função objetivo). Cada elemento selecionado possui um “valor ótimo” ( $O\rho$ , definido na equação 4.1) e o subconjunto dos elementos que minimizam ou maximizam a função objetivo são chamados de “solução ótima” [81].

$$O\rho = \min\{f(x) | x \in T\} \quad (4.1)$$

É possível utilizar soluções probabilísticas, algoritmos de aproximação e/ou heurísticas para resolver problemas de otimização computacional. Entretanto, mesmo com uma quantidade reduzida de casos de teste disponíveis ao testador, o custo computacional para a execução dos testes pode aumentar conforme a quantidade de configurações/opções do *software*. Por exemplo, supondo agora que o *software* descrito anteriormente evolua

para suportar o processamento de imagens com 50 tons de cinza distintos, e que o conjunto de teste disponível seja composto por 100 imagens. Nesse caso, seriam necessárias  $\binom{100}{50} \cong 1 * 10^{19}$  execuções para se testar as 50 opções de cinza utilizando todo o conjunto de teste.

Como se pode observar, selecionar um subconjunto otimizado de casos de teste é um problema fundamental em testes de *software*. O método definido para superar este desafio procura selecionar casos de testes dissimilares entre si e que sejam capazes de encontrar mais rapidamente a presença de erros em programas de processamento de imagens. No contexto do *Tcs&CbIR*, o desafio consiste em encontrar  $n$  casos de teste capazes de 1) otimizar a execução dos testes (o custo computacional de seleção e de execução do subconjunto de teste deve ser muito menor do que o custo da execução de todo o conjunto de teste) e 2) o subconjunto de teste deve maximizar a capacidade de detecção de falhas em relação a todo o conjunto de teste. Portanto, o problema de seleção inerente ao *Tcs&CbIR* pode ser definido como:

*“Dado o programa  $P$ , o conjunto de teste  $T$  e a função de similaridade  $f(x)$ , encontrar um subconjunto de  $T$  ( $sT$ ) para testar  $P$ ”.*

É importante salientar que não é possível definir a capacidade de detecção de falhas do subconjunto  $sT$  antes de sua primeira execução. Todavia, a seleção de casos de teste utilizando conceitos de similaridade tem como base a heurística de que a diversificação dos casos de teste pertencentes ao domínio de entrada tende a maximizar a capacidade de detecção de falhas de  $sT$ . Dessa forma, o problema de seleção pode ser apresentado como uma solução que visa a *minimizar a similaridade média* ( $SimMsr$ )<sup>†</sup> em um espaço de busca multidimensional, conforme apresentado na equação 4.2.

$$SimMsr(sT) = \sum_{CT_i, CT_j \in T \wedge 1 \leq i < j} f(CT_i, CT_j) \quad (4.2)$$

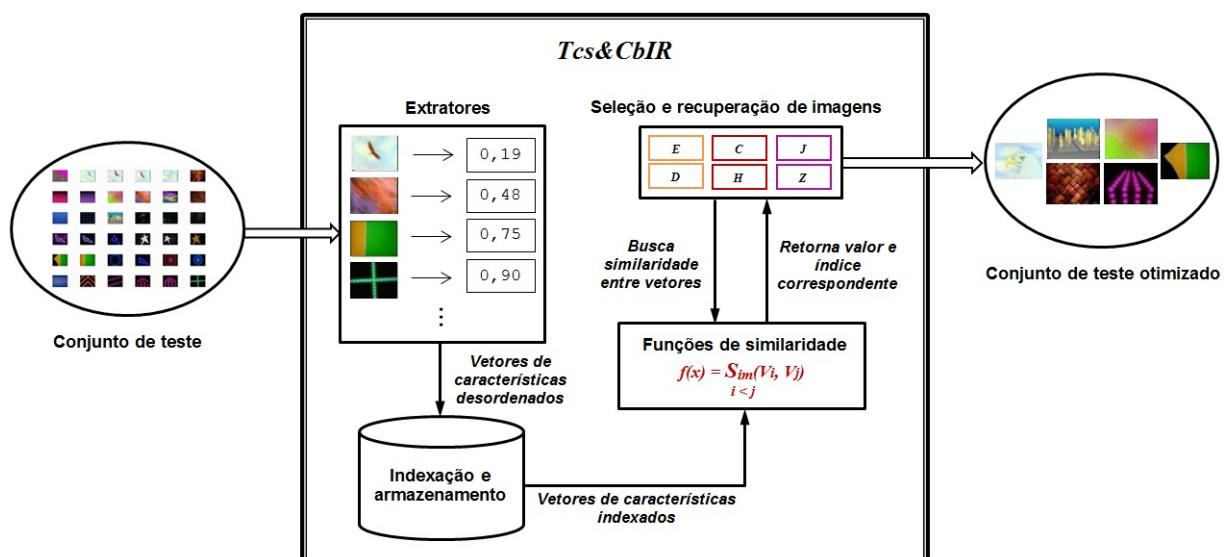
No qual  $f(CT_i, CT_j)$  retorna a distância entre os casos de teste  $CT_i$ ,  $CT_j$  pertencentes ao conjunto  $T$ . Conforme as definições apresentadas verifica-se que o algoritmo de seleção deve estabelecer um padrão de similaridade para escolher os elementos que farão parte do subconjunto  $sT$ , otimizando sua capacidade de detecção de falhas.

---

<sup>†</sup>No contexto apresentado  $SimMsr$  utiliza comparação em pares (*pair-wise comparison*).

## 4.2 Metodologia e definição da proposta - *Tcs&CbIR*

O método *Tcs&CbIR* utiliza conceitos de CBIR para selecionar casos de teste dissími-  
lares considerando a distribuição multidimensional do domínio de entrada. Os casos  
de teste, nesse cenário, são as imagens utilizadas como dados de entrada do *software* sob  
teste (SUT). Para recuperar as imagens mais relevantes, o *Tcs&CbIR* agrega a execução de  
quatro etapas de processamento: 1) extração de características; 2) indexação e armazena-  
mento dos vetores de características; 3) cálculo de similaridade e 4) seleção e recuperação  
de imagens. A Figura 18 ilustra as etapas de processamento do *Tcs&CbIR*.



**Figura 18** – Seleção de casos de teste utilizando conceitos de CBIR

A Figura 18 mostra a seleção de casos de teste utilizando conceitos de CBIR. Dado o conjunto de teste, o método extrai informações das imagens, realiza a indexação e o armazenamento dos vetores de características, calcula a distância entre os vetores e depois seleciona e recupera um subconjunto de casos de teste dissími-  
lares.

No *Tcs&CbIR*, a definição dos extratores e a criação dos vetores de características permitem a aplicação sistematizada de funções de distância para encontrar imagens dis-  
similares em relação às características de interesse.

As estruturas de armazenamento são importantes recursos previstos em sistemas de CBIR e visam a manter um arquivo permanente dos vetores de características. Apesar do custo computacional necessário para a abstração dos dados, os vetores podem ser armazenados em arquivo, evitando o reprocessamento das imagens. Já a indexação consiste em construir estruturas de dados para otimizar o processo de recuperação das imagens.

As etapas de indexação e armazenamento estão diretamente relacionadas à eficiência do *Tcs&CbIR*. Para garantir o bom desempenho do método o desenvolvedor pode criar suas próprias estruturas ou utilizar qualquer estrutura que julgar adequada (e.g. sistema gerenciador de banco de dados).

A escolha dos extractores e das funções de similaridade têm impacto direto na eficácia do *Tcs&CbIR*. A qualidade dos extractores e das funções de distância pode ser mensurada, subjetivamente, em termos de “o quanto bem determinada característica representa a imagem” e “o quanto bem determinada função pode representar a similaridade entre as imagens”. Em termos gerais, o desenvolvedor pode utilizar qualquer extrator de característica (e.g. cor, textura) e função de similaridade (e.g. Euclidiana) que julgar importante.

Na abordagem *Tcs&CbIR* é possível utilizar diversos métodos para agrupar e classificar as imagens com base em suas características, tais como *clustering*, lógica *fuzzy*, algoritmos genéticos, *adaptive random testing*, dentre outros.

Para demonstrar a aplicabilidade do método *Tcs&CbIR* será apresentada, a seguir, a implementação de cada uma das quatro etapas de processamento.

#### 4.2.1 Extractores de características

Delamaro et al. [18] desenvolveram um *framework* que utiliza técnicas de CBIR para a definição de oráculos de teste chamado *O-FIm* (*Oracle for Images*) (vide seção 2.5). O *O-FIm* provê um conjunto de classes desenvolvidas em Java que permite a extração de conteúdos visuais, criação de vetores de características e cálculo de distância entre as imagens. Para implementar o *Tcs&CbIR* foram adaptados três extractores contidos no *O-FIm*: área, cor e perímetro, definidos nos Algoritmos 5, 6 e 7, respectivamente.

```

Entrada: Conjunto de teste composto por imagens
1 para cada imagem do conjunto de teste hacer
2   Largura  $\leftarrow$  largura da imagem
3   Altura  $\leftarrow$  altura da imagem
4   TamanhoImagen  $\leftarrow$  Largura * Altura
5   Cor  $\leftarrow$  qualquer tom de cinza
6   Area  $\leftarrow$  0
7   para cada pixel da imagem hacer
8     se pixel = Cor então
9       Area  $\leftarrow$  Area + 1
10  VetorCaracterísticas [ ]  $\leftarrow$  Area/TamanhoImagen
11 retorna VetorCaracterísticas [ ]
12 Fim da execução

```

**Algoritmo 5:** Extrator de característica “área” para imagens em tons de cinza

**Entrada:** Conjunto de teste composto por imagens

```

1 para cada imagem do conjunto de teste hacer
2   Largura ← largura da imagem
3   Altura ← altura da imagem
4   TamanhoImagen ← Largura * Altura
5   SomaTotal ← 0
6   para cada pixel da imagem hacer
7     Cor ← valor que representa a cor do pixel (e.g. 253 em padrão RGB)
8     SomaTotal ← adicionar o valor de Cor
9   Media = SomaTotal / TamanhoImagen / Quantidade de valores que o pixel pode assumir
(e.g. 256 em padrão RGB)
10  VetorCaracterísticas [ ] ← Media
11 retorna VetorCaracterísticas [ ]
12 Fim da execução

```

**Algoritmo 6:** Extrator de característica “cor” para imagens em padrão RGB

**Entrada:** Conjunto de teste composto por imagens

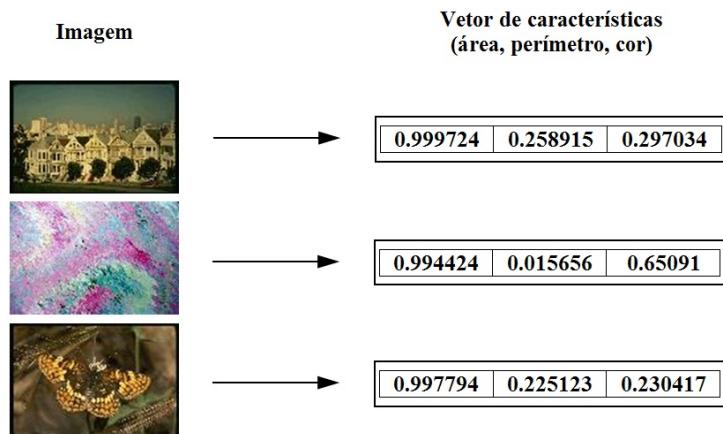
```

1 para cada imagem do conjunto de teste hacer
2   Largura ← largura da imagem
3   Altura ← altura da imagem
4   TamanhoImagen ← Largura * Altura
5   Cor ← preto
6   Perimetro ← 0
7   PerimetroMaximo ← TamanhoImagen
8   Pixels [ ] ← todos os pixels da imagem
9   para i = Largura até TamanhoImagen-Largura fazer
10    se Pixels [posicao i] > Cor e
11      Pixels [posicao i + 1] = Cor ou
12      Pixels [posicao i - 1] = Cor ou
13      Pixels [posicao i + Largura] = Cor ou
14      Pixels [posicao i - Largura] = Cor ou
15      Pixels [posicao i - Largura - 1] = Cor ou
16      Pixels [posicao i - Largura + 1] = Cor ou
17      Pixels [posicao i + Largura - 1] = Cor ou
18      Pixels [posicao i + Largura + 1] = Cor então
19        Perimetro ← Perimetro + 1
20   VetorCaracterísticas [ ] ← Perimetro/PerimetroMaximo
21 retorna VetorCaracterísticas [ ]
22 Fim da execução

```

**Algoritmo 7:** Extrator de característica “perímetro” para imagens binárias

Na implementação do *Tcs&CbIR*, cada característica é representada por um valor numérico cuja escala pode variar entre zero e um. A Figura 19 apresenta um exemplo real de extração das características área, perímetro e cor de imagens.



**Figura 19** – Exemplo real de extração das características área, perímetro e cor

A Figura 19 apresenta três imagens e seus respectivos vetores de características. Nesse exemplo, o extrator de área considera qualquer cor diferente de branco; o extrator de perímetro considera os contornos em cor preta e o extrator de cor calcula a média de cores. A Figura 20 ilustra o resultado de saída da etapa de extração de características do método *Tcs&CbIR*.

Nome da imagem	Caminho	Área	Perímetro	Cor
1.jpg	C:\Tcs&CbIR	0.562498	0.332673	0.425437
80.jpg	C:\Tcs&CbIR	0.954763	0.221456	0.001321
33.jpg	C:\Tcs&CbIR	0.015794	0.854672	0.741369
⋮	⋮	⋮	⋮	⋮
02.jpg	C:\Tcs&CbIR	0.659834	0.264373	0.171891
n.jpg	C:\Tcs&CbIR	0.443895	0.163242	0.718595

**Figura 20** – Resultado de saída da etapa de extração de características

A Figura 20 mostra a composição do vetor de características gerado pelo *Tcs&CbIR* para cada imagem do conjunto de testes. O vetor contém as seguintes informações: 1) nome da imagem; 2) caminho da imagem e 3) valores numéricos que representam as características da imagem.

#### 4.2.2 Indexação e armazenamento

A implementação desta etapa, em termos práticos, consistiu em ordenar e gravar os vetores de características em arquivo, conforme apresentado na Figura 21.

Indexacao-Armazenamento.Tcs&CbIR - Bloco de notas	
Arquivo	Editar
01.jpg; C:\Tcs&CbIR; 0.562498; 0.332673; 0.425437;	...
02.jpg; C:\Tcs&CbIR; 0.659834; 0.264373; 0.171891;	...
...	...
33.jpg; C:\Tcs&CbIR; 0.015794; 0.854672; 0.741369;	...
...	...
80.jpg; C:\Tcs&CbIR; 0.954763; 0.221456; 0.001321;	...
...	...
n.jpg; C:\Tcs&CbIR; 0.443895; 0.163242; 0.718595;	...

**Figura 21** – Indexação e armazenamento dos vetores de características

A Figura 21 ilustra o desenvolvimento do processo de ordenação e armazenamento dos vetores de características. O arquivo de dados (*Indexacao-Armazenamento.Tcs&CbIR*) contém os nomes das imagens ordenados, o caminho para o diretório das imagens e os valores numéricos que representam as características das imagens. Para recuperar as imagens selecionadas a partir do arquivo de dados utilizou-se o algoritmo de pesquisa binária.

#### 4.2.3 Funções de similaridade

O *Tcs&CbIR* utiliza funções de similaridade para atribuir aos casos de teste um número inteiro que representa o seu nível de similaridade. As funções empregadas durante a fase de implementação foram a distância Euclidiana e a distância Canberra, definidas nas equações 4.3 e 4.4, respectivamente.

$$f(x) = \sqrt{2} \sum_{i=0}^{n-1} (a_i - b_i)^2 \quad (4.3)$$

$$f(x) = \sum_{i=0}^{n-1} \frac{|a_i - b_i|}{|a_i| + |b_i|} \quad (4.4)$$

Nas equações apresentadas,  $f(x)$  retorna a distância entre os casos de teste representados por  $a$  e  $b$ . É importante destacar que, visando a diminuir o custo computacional, utilizou-se a comparação em pares para o cálculo de similaridade. O próximo passo consistiu em gerar uma matriz otimizada de tamanho  $(S^2 \div 2) - S$  capaz de representar todas as distâncias possíveis entre os casos de teste. A Figura 22 ilustra a otimização da matriz de similaridade por meio da exclusão de informações redundantes.

$$\begin{array}{c}
 \text{Matriz de similaridade redundante} \\
 \left[ \begin{array}{cccccc}
 A & B & C & D & E & \dots \\
 A & 0 & 3,23 & 15,74 & 13,19 & 6,44 & \dots \\
 B & 3,23 & 0 & 12,53 & 12,04 & 7,50 & \dots \\
 C & 15,74 & 12,53 & 0 & 16,29 & 17,06 & \dots \\
 D & 13,19 & 12,04 & 16,29 & 0 & 19,33 & \dots \\
 E & 6,44 & 7,50 & 17,06 & 19,33 & 0 & \dots \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
 \end{array} \right] \xrightarrow{\hspace{1cm}} 
 \text{Matriz de similaridade otimizada} \\
 \left[ \begin{array}{ccccc}
 A & B & C & D & E \dots \\
 B & 3,23 & & & \\
 C & 15,74 & 12,53 & & \\
 D & 13,19 & 12,04 & 16,29 & \\
 E & 6,44 & 7,50 & 17,06 & 19,33 \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
 \end{array} \right]
 \end{array}$$

**Figura 22 – Otimização da matriz de similaridade**

A Figura 22 mostra que a matriz otimizada não possui informações redundantes devido à característica de simetria das funções de similaridade (por exemplo, a distância entre os dados da *linha A versus coluna B* é a mesma distância entre os dados da *coluna A versus linha B*, e a distância entre os dados da *linha A versus coluna A* é igual a zero).

#### 4.2.4 Seleção e recuperação de imagens

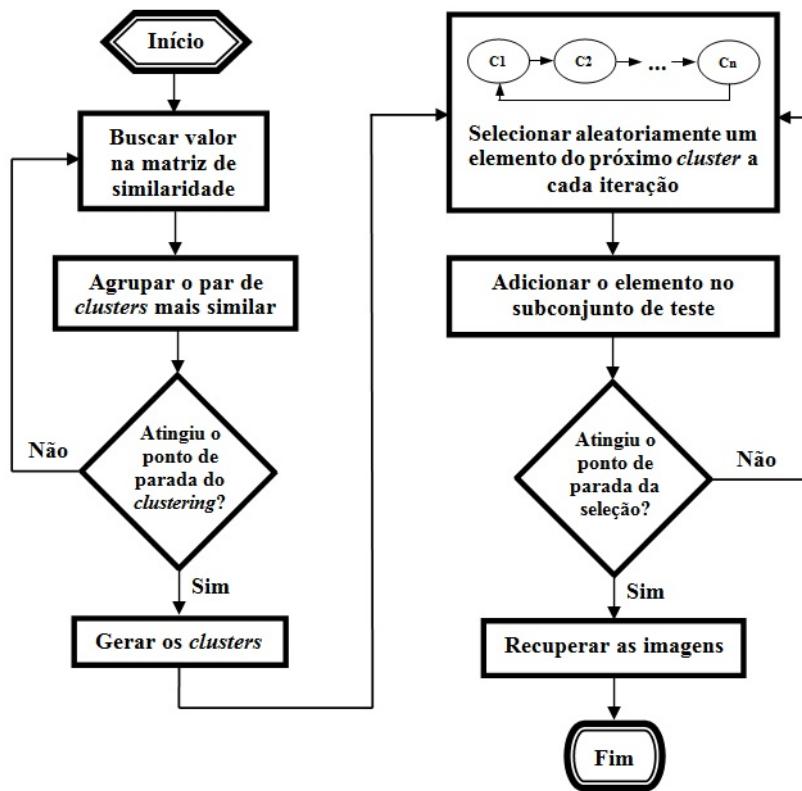
A implementação da presente etapa compreendeu a utilização do método de *clustering* AHC para agrupar os casos de teste similares (vide seção 2.7.1). O AHC utiliza funções de similaridade para dividir o conjunto de teste em *clusters*, de forma que os elementos existentes em um determinado *cluster* sejam similares entre si, porém dissimilares em relação aos elementos de outro *cluster*. O resultado do processo de *clustering* é representado na Figura 23.

<b>Cluster 1</b>	5.jpg	174.jpg	7135.jpg	6523.jpg		
<b>Cluster 2</b>	136.jpg	821.jpg	5555.jpg	1032.jpg	9867.jpg	10000.jpg
<b>Cluster 3</b>	1089.jpg	2645.jpg				
<b>Cluster 4</b>	8543.jpg	100.jpg	3223.jpg	54.jpg	4657.jpg	5423.jpg
<b>Cluster 5</b>	9.jpg	888.jpg	4657.jpg	6665.jpg	3030.jpg	
...	...	...				
<b>Cluster n</b>	222.jpg	74.jpg	8520.jpg			

**Figura 23 – Implementação do clustering AHC**

Os métodos de *clustering* ajudam a identificar o comportamento da distribuição do conjunto de teste através de seu domínio. Tal informação pode ser utilizada para selecionar os casos de teste potencialmente relevantes em termos de detecção de falhas. Por exemplo, *clusters* com elementos similares podem indicar casos de teste redundantes; elementos isolados (*outliers*) indicam condições atípicas cuja existência pode implicar em distorções de interpretação em análises estatísticas (e.g. cálculo de média, mediana, dentre outros).

A Figura 24 apresenta o procedimento completo da etapa de seleção e recuperação de imagens.



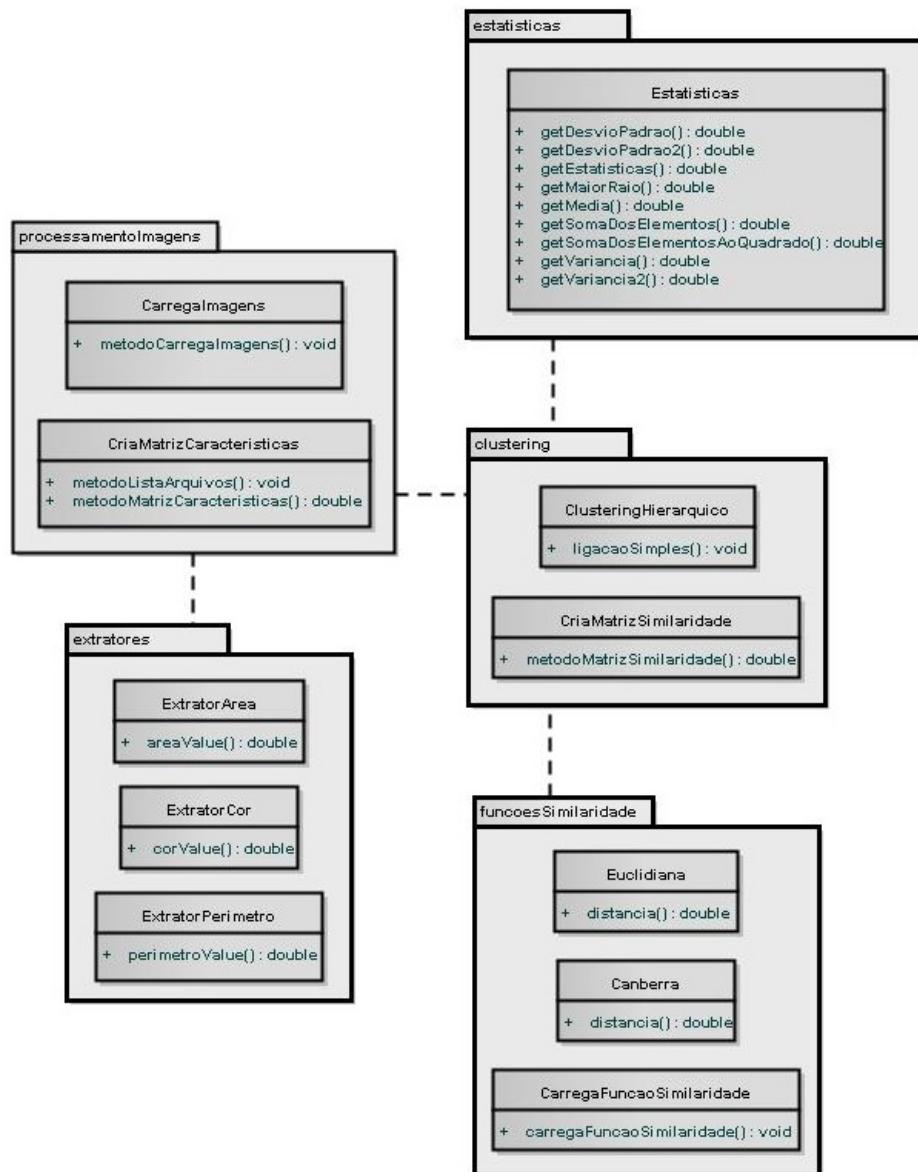
**Figura 24** – Etapa de seleção e recuperação de imagens

A Figura 24 é um fluxograma que descreve a execução da etapa de seleção e recuperação de imagens. Após a geração dos *clusters*, o *Tcs&CbIR* realiza uma seleção aleatória e iterativa tomando um elemento de cada *cluster* (*um-por-cluster*), até que o tamanho do subconjunto de testes desejado seja alcançado.

Para a realização dos testes apresentados no capítulo 6, o ponto de parada da seleção de casos de teste é atingido quando o erro semeado é detectado. Os detalhes sobre o processo de testes e de validação da proposta *Tcs&CbIR* serão discutidos nos próximos capítulos.

#### 4.2.5 Visão geral de implementação

A presente seção descreve o escopo de implementação do método *Tcs&CbIR*. A Figura 25 apresenta o diagrama de pacotes da UML, que representa a estrutura de desenvolvimento do referido método. A implementação dos pacotes também pode ser averiguada no Apêndice C.



**Figura 25 – Estrutura de implementação do Tcs&CbIR**

A Figura 25 mostra a estrutura de implementação do *Tcs&CbIR* por meio de um diagrama de pacotes da UML. Como mencionado na seção 4.2.1, as classes “ExtratorArea”, “ExtratorCor” e “ExtratorPerímetro” foram reutilizadas a partir do *framework O-FIm*, bem como as classes “Euclidiana” e “Canberra”. Após algumas adaptações necessárias<sup>†</sup>, tais classes passaram a representar cerca de 25% de toda a codificação (reuso de 25% de código). Isso significa que, após a implementação dessas classes, ainda foi necessário desenvolver cerca de 75% do código para implementação prática do método *Tcs&CbIR*. A Tabela 10 apresenta a descrição e as principais funcionalidades dos pacotes que compõem o *Tcs&CbIR*.

<sup>†</sup>Um exemplo de adaptação necessária contempla o processo de adaptação do algoritmo de extração de cores para receber imagens em tons de cinza.

**Tabela 10** – Principais funcionalidades dos pacotes que compõem o *Tcs&CbIR*

Pacote	Descrição	Funcionalidade(s)
estatísticas	Fornece funções que retornam cálculos estatísticos	Calcular média, desvio-padrão, quartis, variância, outros
processamentoImagens	Fornece funções de manipulação de imagens	Carregar imagens em memória e criar matriz de características
clustering	Fornece funções de <i>clustering</i>	Criar matriz de similaridade e realizar o <i>clustering</i> hierárquico
extratores	Fornece funções para criar vetores de características	Extrair características de cor, área e/ou perímetro de imagens
funcoesSimilaridade	Fornece funções para calcular a similaridade entre vetores de características	Calcular a distância Euclidiana e/ou Canberra

## 4.3 Considerações finais

Esse capítulo apresentou a definição de uma nova abordagem para seleção de casos de teste utilizando conceitos de CBIR, intitulada *Tcs&CbIR*. Do ponto de vista estrutural, o *Tcs&CbIR* é um método que realiza a recuperação de imagens dissimilares utilizando quatro etapas de processamento. Sob a ótica funcional, o *Tcs&CbIR* pode ser definido como um método que recupera um subconjunto de imagens dissimilares a partir de um vasto conjunto de teste. Assumindo a hipótese de que imagens dissimilares tendem a maximizar a capacidade de detecção de falhas, o subconjunto de teste resultante é um aperfeiçoamento do conjunto de teste original (em termos de tamanho) e pode ser utilizado para a otimização do custo computacional de execução dos testes de *software*.

O próximo capítulo apresenta um plano de teste projetado para avaliar o método *Tcs&CbIR*. Seu objetivo é especificar os *software* que serão testados, como eles serão testados e como o método *Tcs&CbIR* será avaliado.

# *Capítulo 5*

## *Plano de Teste*

O presente capítulo descreve os objetivos gerais e as abordagens de testes de *software* projetados para avaliar a aplicabilidade e a eficácia do método o *Tcs&CbIR*. A seção 5.1 mostra as tecnologias utilizadas para desenvolvimento de *software*; a seção 5.2 apresenta os *software* de processamento de imagens; a seção 5.3 descreve o pré-processamento do conjunto de teste; a seção 5.4 mostra os *software* sob teste; a seção 5.5 apresenta as métricas de avaliação; na seção 5.6 são apresentados os métodos de avaliação dos testes; a seção 5.7 mostra a execução dos testes e saídas esperadas; a seção 5.8 mostra como os extratores de características foram implementados e a seção 5.9 conclui o capítulo.

### **5.1 Tecnologias utilizadas**

No presente trabalho optou-se pelo desenvolvimento dos *software* de processamento de imagens e das funções de *clustering* para se obter melhor controle dos resultados. Tanto o desenvolvimento de *software* quanto do método *Tcs&CbIR* foi realizado utilizando a linguagem de programação Java (jdk 1.7.0-09) [82]. A linguagem Java é gratuita, multi-tarefa, robusta, segura e oferece suporte a aplicações de pequeno, médio e grande porte. Além disso, permite o controle de *threads*, a utilização de bibliotecas de terceiros com encapsulamento e sua plataforma contribui para reutilização de código.

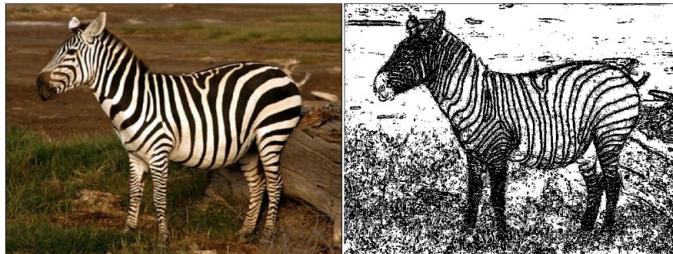
A execução dos testes estatísticos descritos na seção 5.6 foi realizada utilizando o *software* R [83]. O R é uma linguagem e um ambiente de computação estatística para manipulação de dados, cálculos e exibição gráfica, que fornece uma ampla variedade de técnicas matemáticas, além de ser extensível a outras linguagens de programação.

### **5.2 *Software* de processamento de imagens**

Para a implementação dos testes foram desenvolvidos dois *software* de processamento de imagens. Ambos objetivam capturar “eventos” ou mudanças na intensidade de cor em imagens no padrão RGB e realizar transformações a partir dessas mudanças. De fato,

realizar tais transformações pode ser muito importante para a percepção e interpretação de imagens, pois existe a possibilidade de descrever ou de reconstruir uma imagem completa a partir de algumas linhas de borda ou de contorno [11]. Esse tipo de aplicação é comum em sistemas de processamento de imagens, pois as variações de cor podem ser utilizadas para identificar objetos, reconhecer padrões, dentre outros.

O primeiro *software* de processamento de imagens tem o propósito de detectar as bordas de imagens utilizando funções de gradiente, como apresentado na Figura 26.

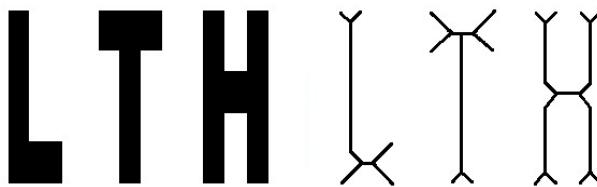


**Figura 26** – Ilustração do processo de detecção de borda utilizando funções de gradiente

A Figura 26 mostra que o *software* em questão captura as mudanças de intensidade de cor mais expressivas da imagem. O resultado é uma imagem binária, na qual os *pixels* das bordas são marcados em preto e os demais *pixels* são marcados em branco. O *software* chama-se *SoftBorda*, e seu código-fonte é apresentado no Apêndice A.

O segundo *software* desenvolvido tem o propósito de *esqueletizar* (ou afinar) imagens construídas em tons de cinza. A esqueletização é uma abordagem importante para a representação de imagens, pois é possível descrever a forma de um objeto não apenas pelo seu perímetro ou área, mas também pelo seu esqueleto.

A esqueletização visa a remover todos os *pixels* redundantes das imagens. O esqueleto pode ser obtido por meio da técnica denominada *transformada do eixo médio*, determinado pelo conjunto de pontos que estão mais próximos da borda de um objeto. Um ponto interno ao objeto pertencerá ao esqueleto se ele possuir, no mínimo, dois *pixels* vizinhos mais próximos da borda [1, 84]. A Figura 27 ilustra o processo de esqueletização.



**Figura 27** – Ilustração do processo de esqueletização

A Figura 27 ilustra o processo de esqueletização de imagens. Ressalta-se que o algoritmo utilizado é capaz de produzir uma nova imagem simplificada na qual o eixo médio possui um único *pixel*. O *software* chama-se *SoftSkeleton*, e seu código-fonte é apresentado no Apêndice B.

### 5.3 Pré-processamento do conjunto de testes

O conjunto de teste original é composto por dez mil imagens coloridas capturadas da *Internet* de forma manual e aleatória, sem qualquer tipo de restrição quanto à sua utilização. Entretanto, existem imagens que contêm dados complexos nas quais os algoritmos de processamento não são capazes de operar adequadamente. Por exemplo, um algoritmo de esqueletização não pode afinar uma imagem que não contenha *pixels* redundantes.

Tendo em vista a potencial generalidade das imagens disponíveis para teste, tornou-se necessário garantir que os *software* de processamento de imagens as executassem de modo eficaz. Foi preciso, portanto, rearranjar o conjunto de testes de acordo com as funcionalidades dos *software* antes de submetê-lo aos testes.

O primeiro passo foi transformar todas as imagens para o padrão RGB, que é o padrão utilizado para o desenvolvimento do *SoftBorda* e do *SoftSkeleton*. O segundo passo foi modificar as imagens do conjunto de testes utilizados no *software* *SoftSkeleton* para tons de cinza, conforme definições apresentadas na seção 5.2. Nesse ponto, há um conjunto de teste composto por imagens coloridas que será executado no *software* *SoftBorda* e outro conjunto de teste que contém as mesmas imagens do conjunto anterior redefinidas em tons de cinza, que será executado no *software* *SoftSkeleton*.

O terceiro e último passo consistiu em avaliar quais imagens os *software* são capazes de processar. Para tanto, os conjuntos de teste foram submetidos aos *SoftBorda* e *SoftSkeleton*. Depois, cada imagem original foi comparada com a sua versão pós-processada *pixel* a *pixel*; as imagens que não apresentaram diferença de resultado em relação às originais foram excluídas do conjunto de teste. Finalmente, restaram quatro mil imagens coloridas no conjunto de teste do *SoftBorda* e sete mil e novecentas imagens em tons de cinza no conjunto de teste do *SoftSkeleton*.

## 5.4 Software sob teste

Para a implementação dos testes foram desenvolvidas versões alternativas dos *software* *SoftBorda* e *SoftSkeleton*: SUT1, SUT2 e SUT3. O SUT1 é uma versão modificada do *SoftBorda* e os SUT2 e SUT3 são versões modificadas do *SoftSkeleton* – cada uma contendo apenas um erro inserido propositalmente (erro semeado). A semeadura de erros não deve ser confundida com os testes de mutação pois, em linhas gerais, os operadores de mutação não são capazes de produzir alguns erros “reais” (em contraste aos erros artificiais), tais como a digitação equivocada de “se variável  $X \geq 10$  então...” ao invés de “se variável  $X \geq 100$  então...” durante o desenvolvimento do *software*.

Visando a aumentar a confiabilidade dos testes foram adotadas duas premissas importantes para a elaboração dos erros semeados: 1) utilizar erros semânticos em detrimento aos erros de escrita de código. Considerou-se que os erros semânticos tendem a ser mais difíceis de serem detectados, já que os erros de escrita de código podem ser encontrados com a ajuda de depuradores automatizados e 2) considerar erros difíceis de serem descobertos aqueles que, durante a fase de testes, sejam detectáveis apenas por uma pequena quantidade de imagens disponíveis nos conjuntos de testes. A semeadura de erros difíceis de serem detectados é importante no contexto de avaliação do método *Tcs&CbIR*, pois quanto menor a quantidade de casos de teste capazes de revelar a presença do erro, maior deve ser a eficácia do método de seleção para encontrá-lo no conjunto de testes. A Tabela 11 apresenta os erros semeados nos SUT1, SUT2 e SUT3.

**Tabela 11** – Erros semeados nos SUT1, SUT2 e SUT3

Software	Linha	Código correto	Erro semeado
SUT1	54	nivelCinza = 255	nivelCinza = 254
SUT2	63	Contador $\geq 6$	Contador $> 6$
SUT3	133	...escalaCinza && p6...	...escalaCinza    p6...

A Tabela 11 mostra os erros semeados nos SUT1, SUT2 e SUT3. A implementação dos erros semeados também pode ser averiguada nos Apêndices A e B.

Dado o conjunto de teste, sua capacidade de detecção dos erros semeados pode ser mensurada por meio do cálculo da quantidade de casos de teste capazes de revelar a presença de erros, conforme procedimento apresentado no Algoritmo 8.

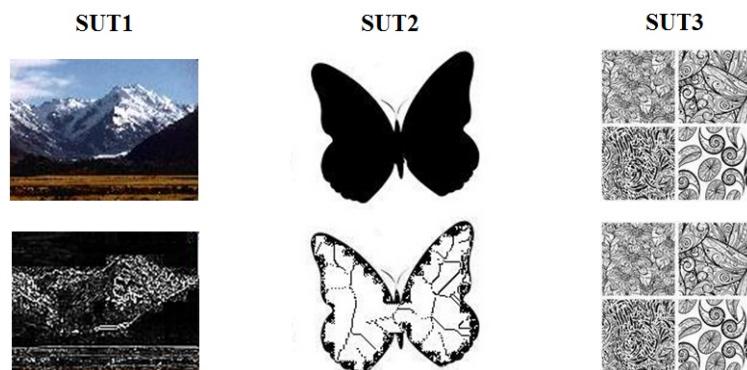
```

Entrada: Conjunto de testes
1 #CT capazes de revelar erros ← 0
2 para cada imagem do conjunto de teste hacer
3   ImagemSC ← execução da imagem no software correto
4   ImagemSUT ← execução da imagem no SUT
5   Comparar ImagemSC e ImagemSUT pixel a pixel
6   se ImagemSC ≠ ImagemSUT então
7     Imprimir “o caso de teste revelou a presença de erro(s)”
8     #CT capazes de revelar erros ← adicionar 1
9   Imprimir “#CT capazes de revelar erros”
10  Fim da execução

```

**Algoritmo 8:** Procedimento para encontrar os casos de teste capazes de revelar a presença de erros

A execução do Algoritmo 8 permitiu verificar a adequação dos erros semeados. O conjunto de teste do SUT1 é composto por quatro mil imagens, das quais apenas trinta e quatro são capazes de revelar a presença do erro (aproximadamente 0,85% do total); o conjunto de teste dos SUT2 e SUT3 é composto por sete mil e novecentas imagens, das quais apenas cinquenta e quarenta e nove, respectivamente, são capazes de revelar a presença do erro (aproximadamente 0,63% do total). A Figura 28 apresenta imagens pré e pós processadas capazes de revelar a presença de erros nos SUT1, SUT2 e SUT3, respectivamente.



**Figura 28 –** Imagens capazes de revelar a presença de erros nos SUT1, SUT2 e SUT3

O resultado obtido após a execução do Algoritmo 8 mostra que, considerando o conjunto de teste disponível, os erros semeados são adequados para a execução dos testes por serem difíceis de se detectar.

## 5.5 Métricas de avaliação

A cobertura de código é uma métrica tradicional e muito utilizada em testes estruturais de *software* (vide seção 3.3.4). Sua importância reside no fato de que, para cada parte de código do SUT deve haver, sempre que possível, pelo menos um caso de teste capaz de exercitá-lo. À medida em que os testes são desenvolvidos, a cobertura destaca os aspectos do código que estão (ou não estão) sendo testados adequadamente [3]. Se a cobertura de código é considerada a principal métrica de qualidade do *software*, seria desejável selecionar um subconjunto reduzido de casos de teste capaz de manter a mesma capacidade de cobertura de código de todo o conjunto de teste.

Entretanto, existem casos em que garantir a cobertura de código não é suficiente para se testar um *software*, pois a detecção de erros depende de como o código é executado. Esse é o caso dos testes abordados nesse trabalho. De fato, a execução de qualquer uma das imagens utilizando o *SoftBorda* e o *SoftSkeleton* apresenta 100% de cobertura de código, mas apenas uma pequena quantidade de imagens consegue revelar a presença do erro semeado. Existem três fatores importantes que podem ajudar a explicar esse fenômeno: 1) os conjuntos de teste disponíveis exigem a execução de todo o código dos SUTs, pois tais conjuntos são compostos por imagens que os SUTs são capazes de processar; 2) o domínio de entrada de *software* que trabalha com imagens é complexo, pois uma única imagem pode conter uma grande quantidade de informações. Por exemplo, uma pequena imagem do tamanho de 100 linhas por 100 colunas contém dez mil *pixels* e 3) os erros semeados são *erros sensíveis aos dados* (em contraste aos *erros sensíveis à cobertura*<sup>§</sup>). Isso significa dizer que, para se detectar a presença desse tipo de erro, é necessário exercitar diferentes comportamentos do *software* sob teste por meio da diversificação dos dados de entrada.

Na literatura considerada (vide capítulo 3) os pesquisadores adotaram diferentes métricas para avaliar a eficácia e a eficiência dos métodos propostos, tais como a taxa de detecção de falhas [44] e a cobertura de modelo ou de código [54]. Os pesquisadores também adotaram o número de casos de teste necessários para encontrar a primeira falha como métrica de avaliação [58, 75]. No contexto de teste proposto, a cobertura de código não mostrou-se uma métrica adequada para avaliar o *Tcs&CbIR*. Torna-se necessário, portanto, adotar métricas alternativas para avaliação.

A métrica utilizada para avaliar a eficácia do *Tcs&CbIR* foi a medida da quantidade de casos de teste necessária para revelar a presença do primeiro erro ( $F_{Tcs\&CbIR}$ ). Em

---

<sup>§</sup>*Erros sensíveis à cobertura* são os erros facilmente detectáveis por meio de cobertura de código.

termos práticos, o  $F_{Tcs\&CbIR}$  é importante porque um erro contido em um determinado ponto do código pode causar outros erros em diferentes pontos do código. Portanto, ao encontrar o primeiro erro, o desenvolvedor deve repará-lo antes de submeter o *software* a um novo teste, evitando assim outros erros resultantes do primeiro [78]. Além disso, essa métrica é particularmente útil em um domínio suscetível a erros sensíveis aos dados, porque casos de teste dissimilares tendem a exercitar diferentes comportamentos do SUT.

O objetivo da metodologia de avaliação proposta é demonstrar que o *Tcs&CbIR* pode ser mais eficaz do que a seleção aleatória, mensurada pela medida *F-measure*<sup>†</sup>. Para demonstrar o aprimoramento do  $F_{Tcs\&CbIR}$  sobre o *F-measure* será utilizada a medida  $F_{rate}$ , expressa em termos de porcentagem. As medidas *F-measure*,  $F_{Tcs\&CbIR}$  e  $F_{rate}$  são definidas nas equações 5.1, 5.2 e 5.3, respectivamente.

$$Fmeasure = \frac{1}{n} \sum_{i=1}^n CT_{aleatorio} \quad (5.1)$$

$$F_{Tcs\&CbIR} = \frac{1}{n} \sum_{i=1}^n CT_{Tcs\&CbIR} \quad (5.2)$$

$$F_{rate} = 1 - \frac{F_{Tcs\&CbIR}}{Fmeasure} * 100\% \quad (5.3)$$

Nas equações 5.1, 5.2 e 5.3  $n$  representa a quantidade de execuções dos testes;  $CT_{aleatorio}$  representa a quantidade de casos de teste necessários para encontrar o primeiro erro em testes aleatórios e  $CT_{Tcs\&CbIR}$  representa a quantidade de casos de teste necessários para encontrar o primeiro erro em testes utilizando o método *Tcs&CbIR*.

## 5.6 Metodologia de avaliação

Tendo em vista que o *Tcs&CbIR* e a seleção aleatória não são determinísticos, os resultados obtidos por meio da execução de diversos testes de *software* serão submetidos a avaliações estatísticas. Enquanto as métricas de avaliação determinam **o quê** e **como** avaliar o *Tcs&CbIR*, os métodos estatísticos visam a revelar o **comportamento** dos testes. Tais avaliações visam a demonstrar que o *Tcs&CbIR* pode ser mais consistente do que a seleção aleatória, isto é, que seus resultados tendem a ser mais eficazes e mais estáveis independentemente da quantidade de execuções do método.

---

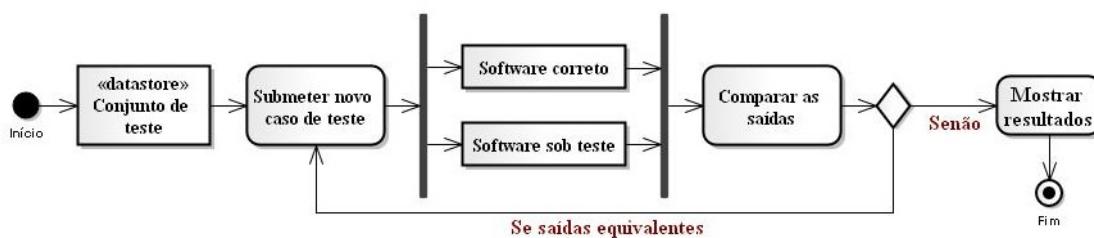
<sup>†</sup>*F-measure* é a quantidade média de casos de teste necessários para encontrar o primeiro erro utilizando a seleção aleatória.

Para tanto, a avaliação dos resultados de seleção de casos de teste foi realizada por meio dos métodos estatísticos apresentados no Apêndice D: cálculo de variabilidade, histogramas de frequência absoluta e *boxplots*. No caso do *Tcs&CbIR* foram utilizados métodos adicionais para avaliar o grau de associação dos três extratores de características disponíveis, como o *scatterplot*, o cálculo de correlação, de qui-quadrado e do *p-valor*.

No contexto de avaliação do *Tcs&CbIR* os seguintes resultados são desejáveis: 1) histogramas com maior densidade à esquerda, conforme ilustrado na Figura 38A; 2) *boxplots* simétricos e sem *outliers*; 3) variabilidade dos resultados minimizada; 4) *scatterplots* que indiquem relação não linear entre os extratores de características; 5) resultados de correlação e de  $\chi^2$  minimizados e 6) *p-valor* menor ou igual a 0,05. Para obter um resultado estatisticamente consistente, os testes de *software* foram executados cem vezes, adotando-se as seguintes medidas como resultado final: média, mediana e desvio-padrão( $\sigma$ ).

## 5.7 Execução dos testes e saídas esperadas

A fase de execução dos testes consiste em selecionar as imagens a partir de um conjunto de teste predefinido e submetê-las tanto ao SUT (que contém erros) quanto ao *software* original (considerado correto). A Figura 29 ilustra o processo de execução dos testes por meio de um diagrama de atividades da UML.



**Figura 29 – Processo de execução dos testes**

A Figura 29 mostra os passos de execução dos testes. Dado o conjunto de teste, executam-se os passos: 1) submeter um novo caso de teste ao *software* correto e ao SUT; 2) as saídas de ambos os processamentos são comparadas *pixel a pixel* e 3) se as saídas forem equivalentes, retornar ao passo 1; entretanto, se as saídas não forem equivalentes, encerrar os testes e exibir em tela o seguinte resultado: “foram necessários  $nCT$  casos de teste para encontrar a presença de erros”, na qual  $nCT$  representa a quantidade de casos de teste submetida ao SUT.

## 5.8 Implementação e pré-avaliação dos extractores de características

Esta seção tece uma apresentação inicial sobre a implementação e a pré-avaliação dos extractores de características e mostra como eles podem influenciar o método *Tcs&CbIR*.

Os extractores de características são elementos determinantes da eficácia do *Tcs&CbIR*. Mesmo com o emprego de funções de similaridade e de técnicas de seleção e recuperação apropriadas, o *Tcs&CbIR* certamente não apresentará bons resultados caso os extractores não sejam capazes de abstrair as informações necessárias de um conjunto de imagens. Em linhas gerais, as imagens podem conter várias características distintas (e.g. cor, forma ou textura) e, portanto, definir bons extractores para um conjunto de imagens genérico tende a ser uma tarefa complexa.

Na seção 4.2.1 foram apresentados três extractores de características distintos: cor, área e perímetro. Para determinar quais extractores deveriam ser utilizados durante a execução dos testes considerou-se, empiricamente, as funcionalidades do SUT1 (*SoftBorda*) e dos SUT2 e SUT3 (*SoftSkeleton*).

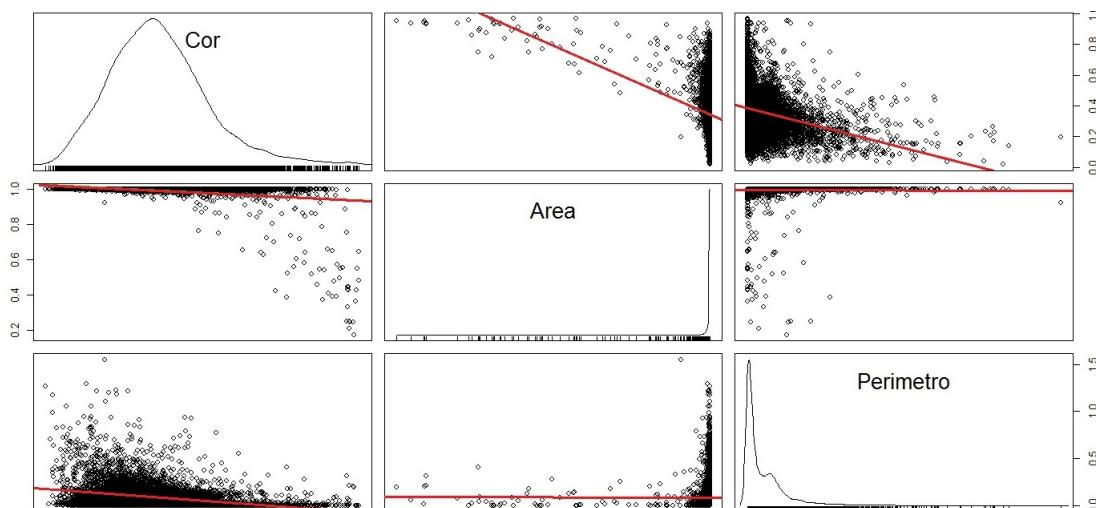
O SUT1 captura as mudanças de cores mais intensas para detectar as bordas de uma imagem. Utilizou-se, portanto, o extrator de “cor” para abstrair as características do conjunto de teste.

Os SUT2 e SUT3 excluem os *pixels* redundantes de imagens em tons de cinza. Inicialmente optou-se pela utilização do extrator “cor” para abstrair as características do conjunto de teste. Entretanto, análises preliminares revelaram que esse extrator é pouco eficiente quando as imagens utilizadas para teste são representadas em tons de cinza. Isso acontece porque no sistema RGB as imagens são representadas por três canais de cores (Red, Green, Blue) e, normalmente, cada canal é definido por um espaço de cores cuja representação numérica pode variar entre 0 e 255 (vide seção 2.2). Dessa forma, o sistema RBG pode representar um total de 16.777.216 cores distintas ( $256 \times 256 \times 256$ ), mas o espaço de cores é drasticamente reduzido para 255 cores distintas quando as imagens são representadas em tons de cinza, nas quais todos os canais de cores possuem o mesmo valor (R=G=B).

Considerando tais análises, optou-se pela inclusão de mais dois extractores de características: área e perímetro. Ressalta-se que o *SoftSkeleton* é sensível a variações de forma das imagens, fato indicativo de que tais extractores tendem a ser importantes para encontrar imagens dissimilares em relação às características contempladas.

Com o propósito de aumentar a confiabilidade dos testes foram analisados indícios estatísticos para sustentar a idéia de que os extractores de características denominados cor, área e perímetro, utilizados em conjunto, são adequados para a abstração das imagens e que tais abstrações são relevantes para encontrar a dissimilaridade entre os casos de teste. Para tanto assumiu-se, empiricamente, a hipótese de que características (também chamadas de variáveis, em termos estatísticos) independentes, não correlacionadas e não lineares sejam preferíveis para revelar a dissimilaridade entre os casos de teste. Considerou-se que variáveis dependentes, fortemente correlacionadas ou lineares tendem a revelar o mesmo padrão de falhas, indesejáveis no contexto dos testes de *software*. Em outras palavras, os testes estatísticos visam a responder as seguintes perguntas: 1) qual a influência de cada variável sobre o cálculo de dissimilaridade? 2) qual o impacto que mudanças em qualquer variável provoca nas demais variáveis?

A partir do cálculo dos valores dos três extractores de características distintos para cada uma das imagens do conjunto, construiu-se um *scatterplot* que mostra a relação linear entre as variáveis. A Figura 30 apresenta o *scatterplot* dos extractores cor, área e perímetro. Os vetores de características foram submetidos ao *software* R [83] que, por sua vez, retornou os resultados dos testes.



**Figura 30 – Scatterplot dos extractores cor, área e perímetro**

A Figura 30 é um *scatterplot* e mostra o comportamento entre os pares de características (cor, área e perímetro). Cada ponto representa um caso de teste e a linha em destaque representa a reta de regressão. Se há muitos pontos seguindo a reta de regressão, há indícios de que as variáveis possuem o mesmo padrão (linear) de comportamento. Portanto, o *scatterplot* apresenta evidências de que o comportamento dos dados (todas as variáveis) é do tipo não-linear.

Adicionalmente foram realizados testes de  $\chi^2$ , de correlação e de *p*-valor para avaliar a influência de cada característica em relação a cálculo de dissimilaridade. A Tabela 12 apresenta os testes de correlação.

**Tabela 12 – Testes de correlação**

<b>Característica</b>	<b>Correlação</b>		
	Cor	Área	Perímetro
Cor	1	0,31	0,29
Área	0,31	1	0,008
Perímetro	0,29	0,008	1

A análise de correlação mostra que as variáveis são fracamente correlacionadas (tendo em vista que os resultados são muito menores que um), indicando que qualquer modificação em uma variável pode causar pouca ou nenhuma mudança de comportamento nas demais variáveis. Os resultados do *p*-valor do teste de  $\chi^2$  são menores que 0,05 e indicam que as variáveis são independentes.

Todos os testes executados sugerem que as variáveis não são lineares, são fracamente correlacionadas, independentes e que todas as variáveis têm forte influência no cálculo de dissimilaridade. Com base nessas estatísticas é possível inferir que a remoção de qualquer variável (cor, área ou perímetro) pode comprometer a eficácia na seleção utilizando a heurística de dissimilaridade nos testes com o *SoftSkeleton*.

É importante ressaltar que os resultados estatísticos representam evidências que dão suporte à hipótese que os extratores de características utilizados são adequados para a execução dos testes.

## 5.9 Considerações finais

Esse capítulo apresentou um plano de teste desenvolvido para avaliar o método *Tcs&CbIR*. Foram especificadas as tecnologias empregadas, os *software* de processamento de imagens e o conjunto de teste que serão utilizados para a execução dos testes. Também foram apresentadas as métricas e os métodos de avaliação dos resultados, bem como a implementação e a pré-avaliação dos extratores de características que serão utilizadas para validar o *Tcs&CbIR*.

Com base na execução dos testes projetados são discutidos, no próximo capítulo, os resultados alcançados e as análises pertinentes ao objeto de pesquisa.

# *Capítulo 6*

## *Resultados e Discussões*

O presente capítulo mostra os resultados obtidos por meio da execução de testes funcionais em *software* de processamento de imagens utilizados para avaliar o método *Tcs&CbIR*. O objetivo é demonstrar que a abordagem proposta pode superar a seleção aleatória, comparando os resultados mensurados pelas medidas *F-measure*,  $F_{Tcs\&CbIR}$  e  $F_{rate}$ . Para tanto, a seção 6.1 apresenta uma visão geral dos testes e as questões de pesquisa; a seção 6.2 mostra a análise e os resultados da execução do processo de *clustering*; a seção 6.3 descreve as análises e os resultados dos testes realizados com os SUT1, SUT2 e SUT3; a seção 6.4 tece a discussão pertinente aos resultados finais e às limitações da pesquisa e a seção 6.5 conclui o capítulo.

### **6.1 Descrição geral dos testes e questões de pesquisa**

A execução do plano de teste apresentado no capítulo 5 tem o objetivo de avaliar o método *Tcs&CbIR*, e suas características gerais podem ser sumarizadas da seguinte maneira:

- **Heurística de teste:** o método *Tcs&CbIR* tem como base a hipótese de que casos de teste dissimilares são propícios para a detecção de erros.
- **Conjuntos de teste:** o conjunto de teste do SUT1 é composto por quatro mil imagens coloridas e o conjunto de teste dos SUT2 e SUT3 é composto por sete mil e novecentas imagens em tons de cinza.
- **Técnicas de teste:** considerando que os testes estruturais (e.g. cobertura de código) não são adequados no contexto de avaliação proposto, optou-se pela inserção de erros semeados como forma de desenvolvimento dos SUTs.
- **Extratores de características:** os extratores disponíveis são cor, área e perímetro.
- **Funções de similaridade:** as funções utilizadas foram a distância Euclidiana e a distância Canberra.

- **Métricas de avaliação:** foi utilizada a medida  $F_{Tcs\&CbIR}$  para representar a quantidade de casos de teste necessários para detectar a presença do erro semeado e a  $F_{rate}$  para representar o aprimoramento do  $F_{Tcs\&CbIR}$  sobre o *F-measure*.
- **Execução dos testes:** a execução dos testes de *software* compreende as seguintes etapas: 1) os SUTs são submetidos aos testes utilizando os casos de teste selecionados pelo método *Tcs&CbIR* e pela seleção aleatória; 2) a execução dos testes é encerrada quando a presença do erro semeado for detectada e 3) os testes são executados cem vezes para se obter um resultado estatisticamente relevante.

Tendo em vista que o *Tcs&CbIR* não é determinístico, foi realizada uma avaliação empírica de sua eficácia utilizando diversos métodos estatísticos, descritos no Apêndice D. O objetivo desta avaliação é responder as seguintes questões de pesquisa: **QP1:** em quais condições o *Tcs&CbIR* pode superar a seleção aleatória? e **QP2:** como as funções de similaridade podem influenciar o *Tcs&CbIR*?

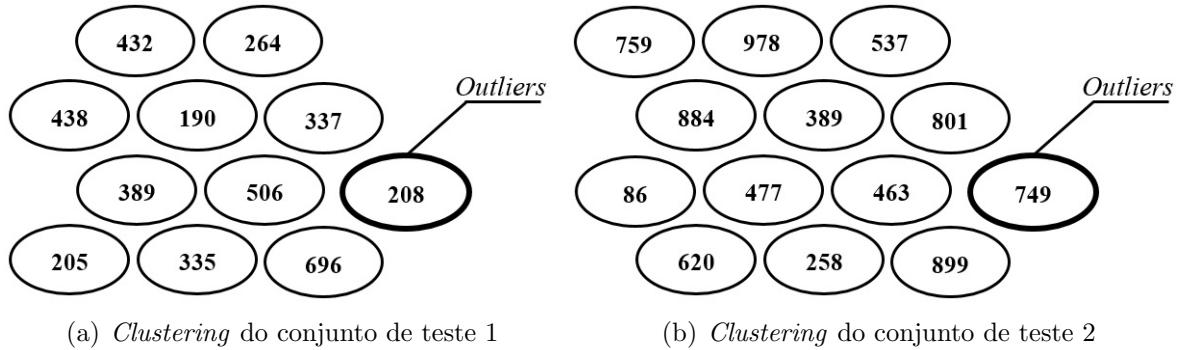
## 6.2 Análise de *clustering*

Essa seção apresenta os resultados e as análises da execução do *clustering AHC*.

Como mencionado na seção 2.7, no método AHC, inicialmente, cada elemento isolado representa um *cluster*. A cada iteração, o algoritmo agrupa o par de *clusters* mais similar, até que o ponto de parada desejado seja alcançado. Nos experimentos apresentados, o ponto de parada do *clustering* foi definido, empiricamente, por meio da análise do nível de fusão. A partir da matriz de similaridade, calculou-se o valor do primeiro quartil, utilizado como parâmetro de parada do *clustering*.

Após a execução do processo de *clustering*, apresentado na seção 4.2.4, obteve-se os seguintes resultados: 1) conjunto de teste do SUT1 com 3736 imagens distribuídas em dez *clusters* e 264 imagens isoladas e 2) conjunto de teste dos SUT2 e SUT3 com 7151 imagens distribuídas em doze *clusters* e 749 imagens isoladas.

As imagens isoladas, em ambos os conjuntos de teste, apresentam valores discrepantes em relação às demais imagens. No contexto dos experimentos apresentados, tais imagens foram consideradas *outliers* e agrupadas em um único *cluster*. A Figura 31 apresenta o resultado do processo de *clustering*.



**Figura 31** – Resultado do processo de clustering

A Figura 31 mostra o resultado do processo de *clustering* para os conjuntos de teste 1 e 2. Após a geração dos *clusters*, o algoritmo implementado no *Tcs&CbIR* realiza a seleção de casos de teste de forma aleatória e iterativa tomando um elemento de cada *cluster* por vez (*um-por-cluster*), até atingir o ponto de parada desejado.

Uma análise mais apurada mostrou que a seleção *um-por-cluster* proporciona ao experimento pelo menos dois comportamentos desejáveis: 1) a similaridade entre os subconjuntos selecionados é minimizada e 2) condições atípicas também podem ser selecionadas, haja vista que o processo de *clustering* naturalmente aloca os *outliers* em *cluster(s)*.

## 6.3 Resultados obtidos

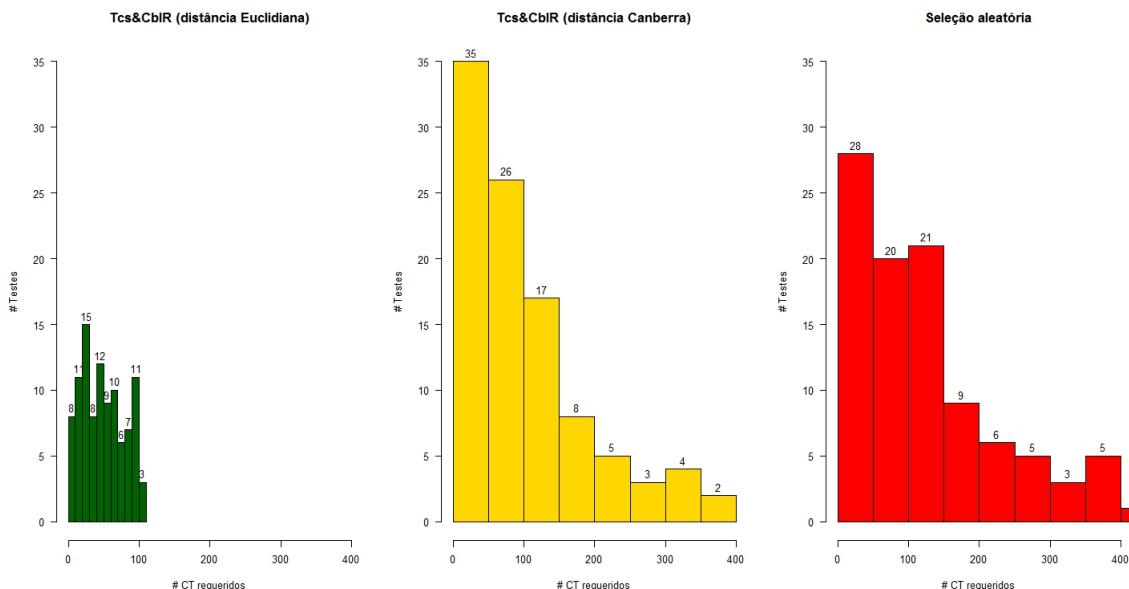
Esta seção apresenta os resultados e as análises dos testes realizados com os SUT1, SUT2 e SUT3. As questões de pesquisa devem ser respondidas por meio da execução dos testes e comparação entre os resultados do *Tcs&CbIR* e da seleção aleatória. A Tabela 13 apresenta os resultados obtidos. A coluna resultado indica a quantidade de casos necessários para encontrar a presença do erro semeado utilizando as medidas *F-measure* e  $F_{Tcs\&CbIR}$ . No caso da medida  $F_{rate}$ , o resultado representa o aprimoramento do conjunto de teste, isto é, o percentual de diminuição da quantidade de casos de teste necessários para encontrar a presença do erro semeado.

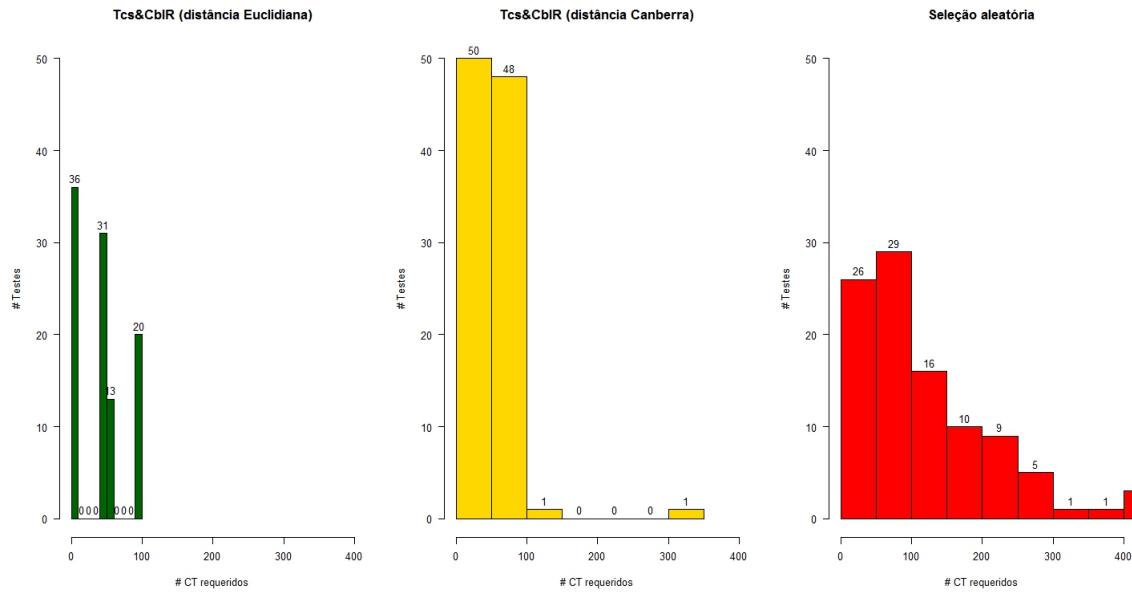
**Tabela 13** – Resultados obtidos utilizando as medidas  $F$ -measure,  $F_{Tcs\&CbIR}$  e  $F_{rate}$ 

<b>Software</b>	<b>Método</b>	<b>Função</b>	<b>Medida</b>	<b>Resultado</b>
SUT1	Aleatório	Não há	$F$ -measure	132
	<i>Tcs&amp;CbIR</i>	Euclidiana	$F_{Tcs\&CbIR}$	<b>49</b>
			$F_{rate}(%)$	<b>63</b>
		Canberra	$F_{Tcs\&CbIR}$	104
			$F_{rate}(%)$	22
SUT2	Aleatório	Não há	$F$ -measure	118
	<i>Tcs&amp;CbIR</i>	Euclidiana	$F_{Tcs\&CbIR}$	<b>44</b>
			$F_{rate}(%)$	<b>63</b>
		Canberra	$F_{Tcs\&CbIR}$	52
			$F_{rate}(%)$	56
SUT3	Aleatório	Não há	$F$ -measure	384
	<i>Tcs&amp;CbIR</i>	Euclidiana	$F_{Tcs\&CbIR}$	131
			$F_{rate}(%)$	66
		Canberra	$F_{Tcs\&CbIR}$	<b>49</b>
			$F_{rate}(%)$	<b>87</b>

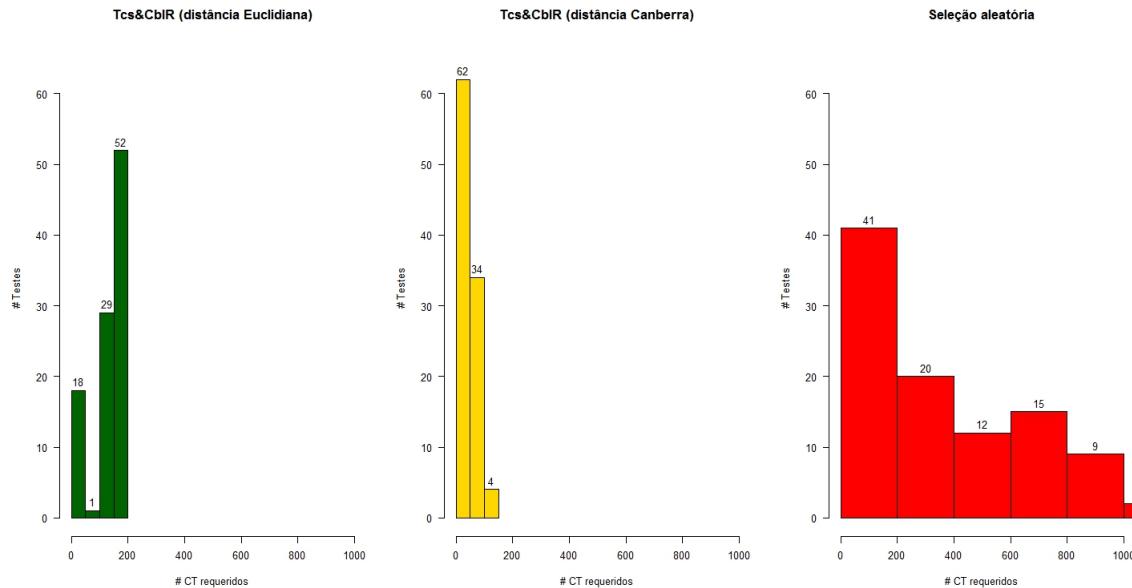
A Tabela 13 mostra que o *Tcs&CbIR* sobrepujou os testes aleatórios. Os melhores resultados, destacados em negrito, apresentam um aprimoramento do conjunto de testes ( $F_{rate}$ ) de 63% para os SUT1 e SUT2 e de 87% para o SUT3. Ao se comparar as distâncias Euclidiana *versus* Canberra, obtém-se as medidas  $F_{rate}$  para o SUT1 na ordem de 63% *versus* 22%; SUT2 63% *versus* 56% e SUT3 66% *versus* 87%, respectivamente.

O próximo passo consistiu em avaliar o comportamento dos resultados obtidos por meio de métodos estatísticos. As Figuras 32, 33 e 34 apresentam os histogramas dos SUT1, SUT2 e SUT3, respectivamente.

**Figura 32** – Histogramas dos testes realizados com o SUT1



**Figura 33 – Histogramas dos testes realizados com o SUT2**



**Figura 34 – Histogramas dos testes realizados com o SUT3**

As Figuras 32, 33 e 34 apresentam os histogramas divididos em classes. O intervalo das classes dos SUT1 e SUT2 são 0–100, 100–200, 200–300 e 300–400 e do SUT3 é 0–200, 200–400, 400–600, 600–800 e 800–1000. O *eixo X* dos histogramas (que representa as classes) mostra a quantidade de casos de teste requeridos para revelar a presença do erro semeado, e o *eixo Y* representa a quantidade de execuções dos testes de *software*.

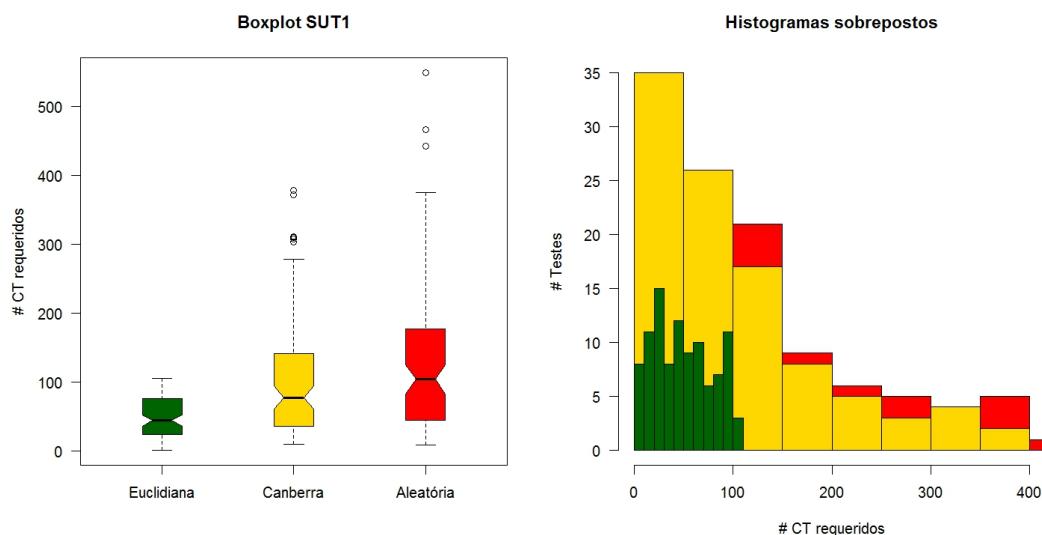
Os histogramas dos SUT1, SUT2 e SUT3 mostram que a distribuição de frequência do *Tcs&CbIR*, quando comparado com a seleção aleatória, apresenta maior densidade

à esquerda. Um número maior de execuções dispostas nos primeiros intervalos do histograma indica que o *Tcs&CbIR* é mais eficaz, pois foram necessários menos casos de teste para revelar a presença do erro semeado. É importante destacar que a maioria dos resultados do *Tcs&CbIR* encontra-se no primeiro intervalo de cada histograma (SUT1 = 0–100, SUT2 = 0–100 e SUT3 = 0–200). A Tabela 14 sumariza os resultados do primeiro intervalo dos histogramas e apresenta os cálculos de variabilidade dos testes de *software*.

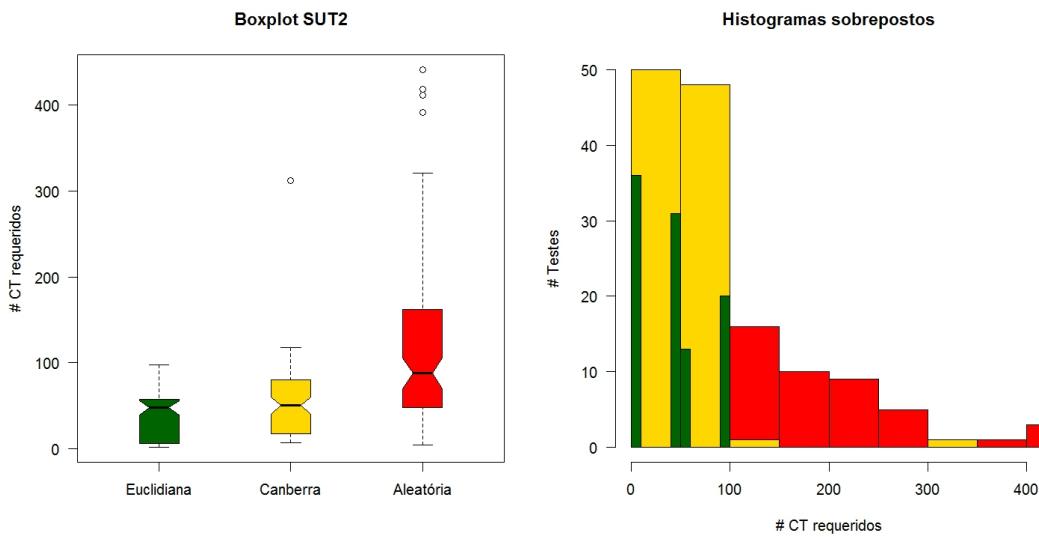
**Tabela 14** – Resultados dos histogramas e dos cálculos de variabilidade

Software	Método	Função	Histograma	Média	Mediana	$\sigma$
SUT1	<b><i>Tcs&amp;CbIR</i></b>	Euclidiana	<b>97</b>	<b>49</b>	<b>44</b>	<b>29</b>
	<i>Tcs&amp;CbIR</i>	Canberra	61	104	77	86
	Aleatório	Não há	48	132	104	115
SUT2	<b><i>Tcs&amp;CbIR</i></b>	Euclidiana	<b>100</b>	<b>44</b>	<b>48</b>	<b>33</b>
	<i>Tcs&amp;CbIR</i>	Canberra	98	52	50	40
	Aleatório	Não há	55	118	88	96
SUT3	<i>Tcs&amp;CbIR</i>	Euclidiana	100	131	163	46
	<b><i>Tcs&amp;CbIR</i></b>	Canberra	<b>100</b>	<b>49</b>	<b>46</b>	<b>28</b>
	Aleatório	Não há	41	384	297	357

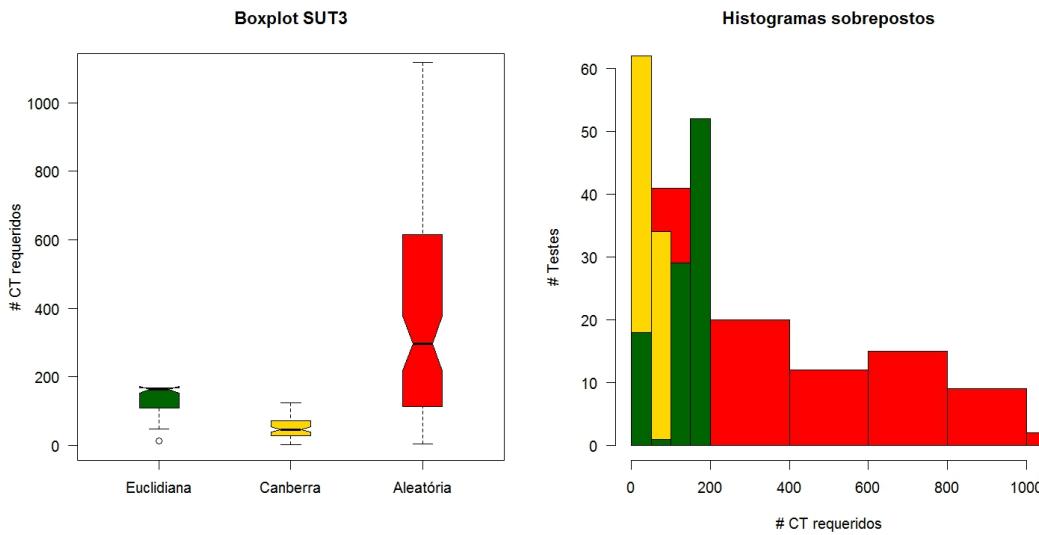
A Tabela 14 mostra que o *Tcs&CbIR* sobrepujou os testes aleatórios em termos estatísticos. Os melhores resultados, destacados em negrito, mostram que a variabilidade do *Tcs&CbIR* é menor em relação a todas as medidas de dispersão empregadas (média, mediana e desvio-padrão). Isso significa que o *Tcs&CbIR* tende a ser mais consistente do que a seleção aleatória. As Figuras 35, 36 e 37 apresentam os *boxplots* e os histogramas sobrepostos dos resultados obtidos.



**Figura 35** – Boxplots e histogramas sobrepostos dos testes realizados com o SUT1



**Figura 36** – Boxplots e histogramas sobrepostos dos testes realizados com o SUT2



**Figura 37** – Boxplots e histogramas sobrepostos dos testes realizados com o SUT3

As Figuras 35, 36 e 37 mostram que os resultados do *Tcs&CbIR* são mais simétricos em relação à seleção aleatória. É possível observar que os boxplots do *Tcs&CbIR* são mais semelhantes entre si, tanto em relação aos limites inferiores/superiores quanto aos quartis. Além disso, a quantidade de *outliers* na seleção aleatória é maior.

Nos histogramas sobrepostos as colunas destacadas em verde indicam melhores resultados utilizando o *Tcs&CbIR* e a distância Euclídea; as colunas destacadas em amarelo indicam melhores resultados utilizando o *Tcs&CbIR* e a distância Canberra e as colunas destacadas em vermelho indicam melhores resultados utilizando a seleção aleatória.

A características particulares de cada função de similaridade empregada neste trabalho podem ser utilizadas para analisar a diferença entre os resultados dos experimentos. A distância Euclidiana pode ser influenciada por qualquer característica que apresente grande variação entre os vetores que estão sendo comparados. Já a distância Canberra normaliza a diferença entre os pares de características equivalentes por meio da soma das características de cada par, e por isso não é tão sensível a grandes diferenças em apenas uma das características. Entretanto, a distância Canberra é mais sensível a pequenas variações [85].

Uma explicação plausível para os resultados alcançados com o SUT1 é que a distância Euclidiana foi capaz de caracterizar a diferença entre as imagens de maneira mais eficaz, possivelmente pelo fato de haver apenas uma única característica sendo considerada para o cálculo de similaridade. No caso do SUT2, é possível que as imagens selecionadas por meio da distância Euclidiana tenham uma relação mais forte com o tipo de erro que foi semeado no *software*. Neste caso, há suspeitas de que o erro semeado seja detectável mais facilmente por meio da execução de imagens com um grau de variabilidade diferente em relação às imagens selecionadas utilizando a distância Canberra.

Ainda é possível inferir, empiricamente, que a distância Canberra apresentou melhores resultados nos experimentos com o SUT3 devido à utilização de múltiplos extratores de características, pois o erro semeado foi mais facilmente detectado devido ao grau de sensibilidade a variações das características empregadas (cor, área e perímetro).

É importante ressaltar que as análises das funções de similaridade foram realizadas por meio de observações. Torna-se necessário, portanto, estudos adicionais que permitam relacionar a utilização das distâncias Euclidiana e Canberra com os erros semeados, bem como a relação destas com os extratores de características e com o conjunto de testes.

Por fim, os resultados mostram que as funções de similaridade podem influenciar diretamente a eficácia do *Tcs&CbIR*. Entretanto, não foram encontradas evidências que permitam determinar a superioridade de qualquer uma das funções de similaridade em relação a outra.

Todos os testes estatísticos demonstram que o *Tcs&CbIR* tende a apresentar melhores resultados independentemente da quantidade de execuções dos testes de *software*. Em resumo, os resultados mostram que 1) o *Tcs&CbIR* foi mais eficaz e mais consistente do que a seleção aleatória em todos os testes realizados (**QP1**) e 2) as funções de similaridade tendem a influenciar fortemente o *Tcs&CbIR*, pois a diferença entre os resultados obtidos utilizando a distância Euclidiana e Canberra são significativos (**QP2**).

## 6.4 Discussões

Na seção anterior foram apresentados estudos experimentais para avaliar a eficácia do método *Tcs&CbIR*. O *Tcs&CbIR* está inserido em um contexto de seleção de casos de teste com base em similaridade (*similarity-based test case selection*), cuja aplicabilidade tem ganhado espaço na comunidade científica graças a diversos estudos que corroboram sua eficiência e eficácia em testes de *software*. Neste cenário, o presente trabalho surge como uma importante contribuição de pesquisa, pois os resultados obtidos mostram *como* a utilização de conceitos de CBIR pode ajudar a reduzir o custo computacional de execução dos testes. No contexto apresentado, o *Tcs&CbIR* superou a seleção aleatória, pois a quantidade de casos de teste necessários para revelar a presença do erro semeado foi reduzida em até 87%.

Os resultados mostram que as funções de similaridade utilizadas exerceram impacto direto na eficácia do *Tcs&CbIR*. A distância Euclidiana apresentou melhores resultados nos testes com os SUT1 e SUT2; já a distância Canberra apresentou melhores resultados nos testes com o SUT3. Verifica-se, portanto, que em determinados experimentos a distância Euclidiana apresentou melhores resultados. Em outros, a distância Canberra foi superior. Devido a essa característica, não foram encontradas evidências que permitam determinar a superioridade de qualquer função de similaridade em relação à outra.

A qualidade dos extractores de características desempenha um papel fundamental no *Tcs&CbIR* (vide seção 5.8). Os resultados confirmam que os extractores de cor, área e perímetro foram adequados para a execução dos testes de *software*. A necessidade da utilização de múltiplos extractores reside no fato de que uma determinada imagem *A* pode conter a mesma característica de outra imagem *B* (área, por exemplo) e, nesses casos, torna-se necessário abstrair informações adicionais para medir a dissimilaridade entre as imagens *A* e *B*. Destaca-se que a abordagem proposta incluiu uma fase de análise estatística dos extractores usados a fim de definir a correlação entre eles e, desta forma, processar o mínimo de informações possível.

Uma limitação dos estudos é o fato de que os *software* sob teste são relativamente pequenos (SUT1 com aproximadamente 100 linhas de código; SUT2 e SUT3 com aproximadamente 140 linhas de código) e que a quantidade de extractores avaliados foi pequena. Entretanto, foram empregadas diferentes metodologias para aumentar a confiabilidade dos testes, tais como: 1) desenvolvimento sistematizado dos extractores de características e dos *software* sob teste e 2) utilização de funções de similaridade bem estabelecidas na

literatura sobre seleção de casos de teste.

As métricas  $F\text{-measure}$ ,  $F_{Tcs\&CbIR}$  e  $F_{rate}$  mostraram-se importantes instrumentos de avaliação do  $Tcs\&CbIR$ . Tais métricas foram capazes de refletir a eficácia dos métodos de seleção abordados pois, evidentemente, quanto menor a quantidade de casos de teste necessários para encontrar a presença do erro semeado, menor o valor assumido por essas medidas.

O método  $Tcs\&CbIR$  utiliza conceitos de CBIR para selecionar casos de teste dissimilares por meio da execução de quatro etapas de processamento: extração de características, indexação e armazenamento dos vetores de características, cálculo de similaridade e seleção e recuperação de imagens. Essa abordagem é flexível e generalista, pois fica a encargo do desenvolvedor escolher as técnicas que julgar importante para implementação de cada uma das etapas. De fato, é possível adaptar o  $Tcs\&CbIR$  ao contexto que se fizer necessário, e sua utilização pode se extender a testes em qualquer tipo de programa de PI.

## 6.5 Considerações finais

O presente capítulo apresentou os resultados e as discussões acerca dos testes utilizando uma nova abordagem de seleção de casos de teste. Foram apresentados estudos que englobam métodos de seleção aleatória e  $Tcs\&CbIR$ , bem como análises estatísticas para ratificar os resultados alcançados. No próximo capítulo são apresentadas as conclusões finais e os trabalhos futuros.

# *Capítulo 7*

## *Conclusões*

Em linhas gerais, os sistemas de CBIR são utilizados com o propósito de encontrar as imagens mais similares em relação a uma imagem de comparação, considerando-se uma ou várias características de interesse. No presente trabalho, este conceito foi adaptado para melhorar a seleção de casos de teste para programas de processamento de imagens. Apesar da existência de vários métodos de seleção de casos de teste bem estabelecidos na literatura, não foram encontradas pesquisas que utilizem conceitos de CBIR neste campo de pesquisa. Visando a preencher tal lacuna, foi apresentado um método inovador para seleção de casos de teste chamado *Tcs&CbIR*. A abordagem proposta utiliza conceitos de CBIR para selecionar e recuperar um subconjunto de imagens capazes de otimizar o custo computacional de execução dos testes de *software*.

Diversos testes realizados em dois programas de processamento de imagens mostram a viabilidade e a eficácia do *Tcs&CbIR*. Os resultados obtidos mostram que o método proposto pode superar a seleção aleatória porque, no contexto de avaliação apresentado, a quantidade de casos de teste necessária para revelar a presença do erro semeado foi reduzida em até 87%. Os resultados revelam, também, o potencial da utilização de CBIR para abstração de informações, bem como a eficácia das funções de similaridade para a seleção de casos de teste.

Apesar do custo de desenvolvimento e de implementação dos extractores, uma grande vantagem proporcionada pelo *Tcs&CbIR* é que a extração de características das imagens só precisa ser realizada uma única vez. De fato, os vetores numéricos que representam as características podem ser armazenados e reutilizados sempre que necessários. Além disso, o *framework O-FIm*, direcionado para a automatização de oráculos de teste, contém diversas classes de extractores e funções de similaridade disponíveis gratuitamente no site do Centro de Competência em *Software Livre* da Universidade de São Paulo<sup>†</sup>.

Outro benefício do *Tcs&CbIR* é a possibilidade de exclusão de imagens redundantes (em termos de características) do conjunto de testes. Após a execução de CBIR e da aplicação das funções de similaridade, é possível manter armazenadas apenas as imagens

---

<sup>†</sup><http://ccsl.icmc.usp.br/pt-br/projects/o-fim-oracle-images>

que atendam a algum critério de dissimilaridade preestabelecido.

A escalabilidade é o terceiro ganho potencial do *Tcs&CbIR*. Sempre há a possibilidade de incluir uma nova imagem no conjunto de testes e, nesse caso, não há a necessidade de reprocessar (via CBIR) as imagens previamente incluídas; basta calcular os extractores para a nova imagem e incluí-los em um banco de dados.

## 7.1 Trabalhos futuros

O *Tcs&CbIR* pode ser explorado em áreas correlatas como a Engenharia de *Software* com base em Busca (*Search-based Software Engineering*). Nesse contexto, a seleção de casos de teste pode ser tratada como um problema de otimização de alta complexidade no qual os métodos *search-based* exploram um espaço de possíveis soluções objetivando encontrar as melhores combinações de resultados com um custo computacional aceitável. Diferentes métodos *search-based* têm sido empregados com sucesso nas mais diversas atividades de teste de *software*, como os algoritmos genéticos e o *particle swarm optimization*. O emprego de métodos *search-based* no contexto do *Tcs&CbIR* pode contribuir para o aumento da confiabilidade dos testes, bem como para a otimização da escalabilidade e da robustez do método.

Existe também a possibilidade de aplicar a nova abordagem no processo de priorização de casos de teste. A priorização ordena os casos de teste sequencialmente e permite que os dados de teste mais importantes sejam executados primeiro. Seu objetivo é encontrar a presença de falhas o mais cedo possível, permitindo uma depuração mais rápida do SUT. Assim como os métodos *search-based*, a priorização tem sido empregada com sucesso principalmente em testes de regressão. Utilizar CBIR e funções de similaridade para priorização de casos de teste é uma possibilidade de pesquisa promissora, haja vista que muitos métodos utilizados para seleção podem ser adaptados para a priorização [86, 87].

A partir dos resultados apresentados no presente trabalho, vislumbram-se os seguintes trabalhos futuros: 1) criar aplicativos para seleção de casos de teste utilizando conceitos de CBIR; 2) desenvolver novos extractores de características que atendam a diferentes necessidades visando a corroborar os resultados alcançados e 3) expandir os testes em termos de tamanho e complexidade de *software*, pois replicar os estudos utilizando novas alternativas pode revelar diferentes padrões de efetividade e eficiência do *Tcs&CbIR*.

## 7.2 Publicações

Esta seção apresenta as publicações geradas até o presente momento.

1. NARCISO, Everton Note; DELAMARO, Márcio Eduardo; NUNES, Fátima de Lourdes dos Santos. Test Case Selection Using CBIR and Clustering. Proceedings of the Nineteenth Americas Conference on Information Systems, Chicago, Illinois, August, 2013.
2. NARCISO, Everton Note; NUNES, Fátima de Lourdes dos Santos; DELAMARO, Márcio Eduardo. Seleção de Casos de Teste Utilizando Conceitos de Variabilidade: Uma Revisão Sistemática. VIII Simpósio Brasileiro de Sistemas de Informação (SBSI 2012).
3. NARCISO, Everton Note; DELAMARO, Márcio Eduardo; NUNES, Fátima de Lourdes dos Santos. Test Case Selection: a Systematic Literature Review. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), August, 2013. [Artigo submetido].
4. NARCISO, Everton Note; DELAMARO, Márcio Eduardo; NUNES, Fátima de Lourdes dos Santos. Test Case Selection Using CBIR Concepts. [Artigo em elaboração].

## *Referências*

- 1 PEDRINI, H.; SCHWARTZ, W. R. Análise de imagens digitais: princípios, algoritmos e aplicações. Primeira Edição. São Paulo, Thomson Learning, 2008.
- 2 NUNES, F. L. S. Introdução ao processamento de imagens médicas para auxílio ao diagnóstico. Atualizações em Informática, v.1, Rio de Janeiro, p.73-126, 2006.
- 3 DELAMARO, M. E.; JINO, M.; MALDONADO, J. C. Introdução ao teste de software. Primeira Edição. Rio de Janeiro, Elsevier, 2007.
- 4 KASURINEN, J.; TAIPALE, O.; SMOLEER, K. Test case selection and prioritization: risk-based or design-based? In Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, p.1-10, 2010.
- 5 SOMMERVILLE, I. Engenharia de software. Oitava Edição. São Paulo, Pearson Addison Wesley, 2007.
- 6 SAPNA, P. G.; MOHANTY, H. Clustering test cases to achieve effective test selection. In Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India, p.1-8, 2010.
- 7 BERTOLINO, A. Software testing research: achievements, challenges, dreams. In Proceedings of Future of Software Engineering, p.85-103, 2007.
- 8 CONCI, A.; AZEVEDO, E.; LETA, F. R. Computação gráfica: teoria e prática, volume 2. Rio de Janeiro, Elsevier, 2008.
- 9 GONZALES, RAFAEL S.; WINTZ, PAUL. Digital image processing. Addison-Wesley Publishing Company, USA, 1987.
- 10 PRAJAPATI, H. B.; VIJ, S. K. Analytical study of parallel and distributed image processing. Image Information Processing. International Conference on, p.1-6, 2011.
- 11 BURGER, W.; BURGE, MARK J. Digital image processing - An algorithmic introduction using Java. Springer Science, USA, 2008.
- 12 P. SAHA; S. SAWARKAR; S. GHOSH. Content-based and association-based image retrieval techniques. In Proceedings of the International Conference and Workshop on Emerging Trends in Technology, p.15-18, 2011.
- 13 C. PATIL; V. DALAL. Content-based image retrieval using combined features. In Proceedings of the International Conference and Workshop on Emerging Trends in Technology, p.102-105, 2011.

- 14 MARQUES, F. L. S. N. Processamento gráfico para aplicações em saúde: técnicas, requisitos, ferramentas, desafios e oportunidades. Texto de Livre Docência. Escola de Artes, Ciências e Humanidades da Universidade de São Paulo (EACH-USP), 2012.
- 15 DATTA, R.; JOSHI, D.; LI, J.; WANG, J. Z. Image retrieval: ideas, influences and trends of the new age. *ACM Computer Surveys*, volume 40, n.2, p.160, 2008.
- 16 OTÁVIO A. B. P.; TORRES, R. S. Eva: an evaluation tool for comparing descriptors in content-based image retrieval tasks. In *Proceedings of the international conference on multimedia information retrieval*, p.413-416, 2010.
- 17 OLIVEIRA, R. A. P.; DELAMARO, M. E.; NUNES, F. L. S. O-FIm – Oracle for Images. *XXIII Simpósio Brasileiro de Engenharia Software*. Fortaleza, Ceará. Anais do XXIII Simpósio Brasileiro de Engenharia de Software, 2009.
- 18 DELAMARO, M. E.; NUNES, F. L. S.; OLIVEIRA, R. A. P. Using concepts of content-based image retrieval to implement graphical testing oracles. *Software Testing, Verification and Reliability*. John Wiley & Sons, volume 23, p.171-198, 2013.
- 19 GONCALVES, V. M.; NUNES, F. L. S.; DELAMARO, M. E. Avaliação de funções de similaridade em sistemas de CBIR: uma revisão sistemática. *Anais do VI Workshop de Visão Computacional*, v.1, FCT-UNESP. Presidente Prudente(SP), p.199-204, 2010.
- 20 MYERS, GLENFORD J. *The art of software testing*. Segunda Edição. New Jersey, EUA. John Wiley & Sons, 2004.
- 21 CARTAXO, E. G.; MACHADO, P. D. L.; NETO, F. G. O. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing Verification and Reliability*, p.75-100, 2011.
- 22 ARCURI, A.; IQBAL, M. Z.; BRIAND, L. Black-box system testing of real-time embedded systems using random and search-based testing. *Lecture Notes in Computer Science*, p.95-110, 2010.
- 23 IEEE STANDARD GLOSSARY OF SOFTWARE ENGINEERING TERMINOLOGY (610.12-1990). Disponível em: <http://standards.ieee.org/findstds/standard/610.12-1990.html>.
- 24 YUE JIA; HARMAN, M. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, p.649-678, 2011.
- 25 PATTON, R. *Software Testing*. Sams Publishing, Indiana, USA, 2001.
- 26 PRESSMAN, R. S. *Software engineering, a practitioner's approach*. Seventh Edition. New York, EUA. McGraw-Hill, 2010.
- 27 MILLS, H. D. On statistical validation of computer programs. *IBM Reports*. FSC72-6015, Federal Systems Division, IBM, Gaithersburg, 1972.
- 28 ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, verification and testing of computer software. *ACM Computing Surveys*, Vol. 14, n2, p.159-192, 1982.

- 29 HUMMEL, O.; ATKINSON, C. Automated harvesting of test oracles for reliability testing. In Proceedings of the 29th Annual International Computer Software and Applications Conference, vol.1, p.196-202, 2005.
- 30 PAULO M. S. B.; JINO, M.; WONG, W. E. Diversity oriented test data generation using metaheuristic search techniques. *Information Sciences*, 2011. Disponível em <http://dx.doi.org/10.1016/j.ins.2011.01.025>.
- 31 NUNES, F. L. S.; DELAMARO, M. E. Recuperação de imagens baseada em conteúdo e sua aplicação na área de saúde. In *Computer on the Beach - Livro de minicursos*, v.1, 1ed. Florianópolis, p.115-144, 2010.
- 32 JOHNSON, R.; FOOTE, B. Designing reusable classes. In *Journal Of Object-Oriented Programming*, Vol.1, Number 2, p.22-35, 1988.
- 33 GUERRA, E.; YODER, J. An evolutionary vision of framework development. *Anais do VIII Simpósio Brasileiro de Sistemas de Informação*. Sociedade Brasileira de Computação, p. 19–24. São Paulo, 2012.
- 34 CENTRO DE COMPETÊNCIA EM SOFTWARE LIVRE DA UNIVERSIDADE DE SÃO PAULO. Disponível em: <http://ccsl.icmc.usp.br/pt-br/projects/o-fim-oracle-images>.
- 35 HEMMATI, H.; BRIAND, L.; ARURI, A.; ALI, S. An enhanced test case selection approach for model-based testing: an industrial case study. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, p.267-276, 2010.
- 36 LEDRU, Y.; VEGA, G. T.; BOUSQUET, L. Test suite selection based on traceability annotations. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, p.342-345, 2012.
- 37 ALI, S.; HEMMATI, H; HOLT, N. E.; ARISHOLM, E.; BRIAND, L. Model transformations as a strategy to automate model-based testing: a tool and industrial case studies. *Simula Research Laboratory, Technical Report(2010-01)*, 2010.
- 38 YOO, S; HARMAN, M. Regression testing minimization, selection and prioritization: A survey. *Software Testing Verification and Reliability*, p.67-120, 2012.
- 39 QU, X.; ACHARYA, M.; ROBINSON, B. Impact analysis of configuration changes for test case selection. *Software Reliability Engineering. IEEE 22nd International Symposium on*, p.140-149, 2011.
- 40 ZENG, F.; LI, L.; LI, J.; WANG, X. Research on test suite reduction using attribute relevance analysis. *Eighth IEEE/ACIS International Conference on*, p.961-966, 2009.
- 41 ENSAN, A.; BAGHERI, E.; ASADI, M.; GASEVIC, D.; BILETSKIY, Y. Goal-oriented test case selection and prioritization for product line feature models. *Information Technology: New Generations. Eighth International Conference on*, p. 291-298, 2011.

- 42 HEMMATI, H.; BRIAND, L. An industrial investigation of similarity measures for model-based test case selection. Software Reliability Engineering. IEEE 21st International Symposium on, p.141-150, 2010.
- 43 LI, Z.; HARMAN, M.; HIERONS, R. M. Search algorithms for regression test case prioritization. Software Engineering, IEEE Transactions on, p.225-237, 2007.
- 44 HEMMATI, H.; ARCURI, A.; BRIAND, L. Reducing the cost of model-based testing through test case diversity. In Proceedings of the 22nd IFIP WG international conference on Testing software and systems, p.63-78, 2010.
- 45 GOLDSCHMIDT, R.; PASSOS, E. Data mining, um guia prático. Rio de Janeiro, Elsevier, 2005.
- 46 MINGOTI, S. A. Análise de dados através de métodos de estatística multivariada: uma abordagem aplicada. Primeira Edição. Belo Horizonte, Editora UFMG, 2005.
- 47 DEVORE, JAY L. Probability and statistics for engineering and the sciences. Boston, USA. Brooks/Cole, 2012.
- 48 DeGROOT, MORRIS H.; SCHERVISH, MARK J. Probability and statistics. Boston, USA. Addison-Wesley, 2012.
- 49 BIOLCHINI, J. C. A.; MIAN, P. G.; NATALI, A. C. C.; CONTE, T. U.; TRAVASSOS, G. H. Scientific research ontology to support systematic review in software engineering. Advanced Engineering Informatics, p.133-151, 2007.
- 50 BRERETON, P.; KITCHENHAM, B. A.; BUDGEN, D.; TURNER, M.; KHALIL, M. Lessons from applying the systematic literature review process within the software engineering domain. Journal of Systems and Software, p.571-583, 2007.
- 51 FOULDS, L. R. Combinatorial optimization for undergraduates. Springer-Verlag, New York, 1984.
- 52 HARMAN, M. The current state and future of search based software engineering. In Proceedings of Future of Software Engineering, p.342-357, 2007.
- 53 YOO, S.; HARMAN, M. Pareto efficient multi-objective test case selection. In Proceedings of the international symposium on Software testing and analysis, p.140-150, 2007.
- 54 MIRARAB, S.; AKHLAGHI, S.; TAHVILDARI, L. Size-constrained regression test case selection using multicriteria optimization. Software Engineering, IEEE Transactions on, p.936-956, 2012.
- 55 SOUZA, L. S.; MIREA, P. B. C. DE; PRUDENCIO, R. B. C.; BARROS, F. A. A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort. Tools with Artificial Intelligence. IEEE International Conference on, p.245-252, 2011.
- 56 TSAI, W.T.; ZHOU, X.; PAUL, R. A.; CHEN, Y.; BAI, X. A coverage relationship model for test case selection and ranking for multi-version software. High Assurance Systems Engineering Symposium, p.105-112, 2007.

- 57 MALA, D. J.; MOHAN, V. Quality improvement and optimization of test cases: a hybrid genetic algorithm based approach. SIGSOFT Software Engineering Notes 35, p.1-14, 2010.
- 58 CHEN, T. Y.; DE HAO HUANG; F. C. KUO. Adaptive random testing by balancing. In Proceedings of the 2nd international workshop on random testing: 22nd IEEE/ACM International Conference on Automated Software Engineering, p.2-9, 2007.
- 59 CHEN, T. Y.; KUO, F. C.; LIU, H. Distributing test cases more evenly in adaptive random testing. Journal of Systems and Software, p.2146-2162, 2008.
- 60 CHEN, T. Y.; KUO, F. C.; LIU, H. Adaptive random testing based on distribution metrics. Journal of Systems and Software, p.1419-1433, 2009.
- 61 KUO, F. C.; CHEN, T. Y.; LIU, H.; CHAN, W. K. Enhancing adaptive random testing in high dimensional input domains. In Proceedings of the ACM symposium on Applied computing, p.1467-1472, 2007.
- 62 HEMMATI, H.; ARCURI, A.; BRIAND, L. Empirical investigation of the effects of test suite properties on similarity-based test case selection. Software Testing, Verification and Validation. IEEE Fourth International Conference on, p.327-336, 2011.
- 63 MAYER, J. Towards effective adaptive random testing for higher-dimensional input domains. In Proceedings of the 8th annual conference on genetic and evolutionary computation, p.1955-1956, 2006.
- 64 MAYER, J. Adaptive random testing with randomly translated failure region. In Proceedings of the 1st international workshop on random testing, p.70-77, 2006.
- 65 MAYER, J.; CHEN, T. Y. ; HUANG, DE H. Adaptive random testing through iterative partitioning revisited. In Proceedings of the 3rd international workshop on software quality assurance, p.22-29, 2006.
- 66 ZHI QUAN ZHOU. Using coverage information to guide test case selection in adaptive random testing. Computer Software and Applications Conference Workshops. IEEE 34th Annual, p.208-213, 2010.
- 67 SHIN, S. H; PARK, S. K.; CHOI, K. H.; JUNG, K. H. Normalized adaptive random test for integration tests. Computer Software and Applications Conference Workshops. IEEE 34th Annual, p.335-340, 2010.
- 68 DE LUCIA, A.; DI PENTA, M.; OLIVETO, R.; PANICHELLA, A. On the role of diversity measures for multi-objective test case selection. Automation of Software Test. International Workshop on, p.145-151, 2012.
- 69 BISWAS, S.; MALL, R.; SATPATHY, M.; SUKUMARAN, S. A model-based regression test selection approach for embedded applications. ACM SIGSOFT Software Engineering Notes 34, p.1-9, 2009.
- 70 YU, L.; XU, L.; TSAI, W. Time-constrained test selection for regression testing. In Proceedings of the 6th international conference on Advanced data mining and applications, p.221-232, 2010.

- 71 XU, Z.; GAO, K.; KHOSHGOFTAAR, T. M.; SELIYA, N. System regression test planning with a fuzzy expert system. *Information Sciences*, 2010. Disponível em <http://dx.doi.org/10.1016/j.ins.2010.09.012>.
- 72 CALIEBE, P.; HERPEL, T.; GERMAN, R. Dependency-based test case selection and prioritization in embedded systems. *Software Testing, Verification and Validation. IEEE Fifth International Conference on*, p.731-735, 2012.
- 73 KUMAR, A.; GOEL, R. Event driven test case selection for regression testing web applications. *Advances in Engineering, Science and Management (ICAESM). International Conference on*, p.121-127, 2012.
- 74 WOJCICKI, M. A.; STROOPER, P. An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies. *Software Quality, Fifth International Workshop on*, p.20-26, 2007.
- 75 CHEN, T. Y.; KUO, F. C; MERKEL, R. G.; TSE, T. H. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, p.60-66, 2010.
- 76 ENGSTROM, E.; SKOGLUND, M.; RUNESON, P. Empirical evaluations of regression test selection techniques: a systematic review. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, p.22-31, 2008.
- 77 ITURBIDE, J. ÁNGEL VELÁZQUEZ. The design and coding of greedy algorithms revisited. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, p.8-12, 2011.
- 78 ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, vol.27, p.929-948, 2001.
- 79 FREE SOFTWARE FOUNDATION. Disponível em: <http://www.fsf.org/>.
- 80 SOFTWARE-ARTIFACT INFRASTRUCTURE REPOSITORY (SIR). Disponível em: <http://sir.unl.edu/portal/index.php>.
- 81 HOCHBAUM, D. S. Approximation algorithms for NP-Hard problems. PWS Publishing, Boston, 1997.
- 82 SITE OFICIAL DO JAVA. Disponível em [http://www.java.com/pt\\_BR/about/](http://www.java.com/pt_BR/about/).
- 83 R CORE TEAM. R: A language and environment for statistical computing. R Foundation for Statistical Computing. Vienna, Austria. ISBN 3-900051-07-0, 2012. Disponível em: <http://www.R-project.org/>.
- 84 NIXON, M.; AGUADO, A. Feature extraction & image processing. Oxford, UK. Elsevier, 2008.
- 85 BUGATTI, P. H.; TRAINA, A. J. M.; TRAINA-Jr., C. Assessing the best integration between distance-function and image-feature to answer similarity queries. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, p.1225-1230, 2008.

- 86 BO, J.; ZHANG, Z.; CHAN, W. K.; TSE, T. H. Adaptive random test case prioritization. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, p.233-244, 2009.
- 87 CATAL, C. On the application of genetic algorithms for test case prioritization: a systematic literature review. In Proceedings of the 2nd international workshop on Evidential assessment of software technologies, p.9-14, 2012.

# *APÊNDICE A*

## *Código-Fonte do SoftBorda*

```

1 package processamentoImagens; //SoftBorda
2 import java.awt.image.BufferedImage;
3 import java.awt.image.Raster;
4 import java.awt.image.WritableRaster;
5 import java.io.File;
6 import java.io.IOException;
7 import java.io.FilenameFilter;
8 import java.math.BigDecimal;
9 import javax.imageio.ImageIO;
10 public class Gradiente {
11     public static void main(String[] args) throws IOException {
12         Gradiente.metodoGradiente("C:\\\\Banco de imagens \\\\", \"C:\\\\Gradiente\\\\");
13     }
14     public static void metodoGradiente(String path, String salvaImagens) throws IOException {
15         int linha, coluna, nivelCinza, nivelCinzaVizinhoLin, nivelCinzaVizinhoCol, qtdImagens;
16         int i, j, banda, bandas, largura, altura, contadorNome = 0, red, green, blue;
17         int[] vCor;
18         String arquivo;
19         File[] imagens;
20         CarregaImagens carrega = new CarregaImagens();
21         imagens = carrega.metodoCarregaImagens(path);
22         qtdImagens = imagens.length;
23         for (j = 0; j < qtdImagens; j++) {
24             try {
25                 BufferedImage imagem = ImageIO.read(imagens[j]);
26                 Raster rasterLeitura = imagem.getRaster();
27                 bandas = rasterLeitura.getNumBands();
28                 if (bandas != 3) {
29                     continue;
30                 }

```

```

31     largura = imagem.getWidth();
32     altura = imagem.getHeight();
33     WritableRaster rasterEscrita = imagem.getRaster();
34     vCor = new int[bandas];
35     for (linha = 0; linha < altura - 1; linha++) {
36         for (coluna = 0; coluna < largura - 1; coluna++) {
37             rasterEscrita.getPixel(coluna, linha, vCor);
38             red = vCor[0];
39             green = vCor[1];
40             blue = vCor[2];
41             nivelCinza = new BigDecimal((red + green + blue) / 3).setScale(0,
42                                         BigDecimal.ROUND_HALF_UP).intValue();
42             rasterEscrita.getPixel(coluna, linha + 1, vCor);
43             red = vCor[0];
44             green = vCor[1];
45             blue = vCor[2];
46             nivelCinzaVizinhoLin = new BigDecimal((red + green + blue) / 3).setScale(0,
47                                         BigDecimal.ROUND_HALF_UP).intValue();
47             rasterEscrita.getPixel(coluna + 1, linha, vCor);
48             red = vCor[0];
49             green = vCor[1];
50             blue = vCor[2];
51             nivelCinzaVizinhoCol = new BigDecimal((red + green + blue) / 3).setScale(0,
52                                         BigDecimal.ROUND_HALF_UP).intValue();
52             nivelCinza = Math.abs(nivelCinza - nivelCinzaVizinhoLin) + Math.abs(nivelCinza -
53                                         nivelCinzaVizinhoCol);
53             if (nivelCinza > 255) {
54                 nivelCinza = 255; // -Erro semeado: “nivelCinza = 254”.
55             }
56             for (i = 0; i < bandas; i++) {
57                 rasterEscrita.setSample(coluna, linha, i, nivelCinza);
58             }
59         }
60     }
61     for (linha = 0; linha < altura - 1; linha++) {
62         for (coluna = 0; coluna < largura - 1; coluna++) {
63             for (banda = 0; banda < bandas; banda++) {
64                 rasterEscrita.getPixel(coluna, linha, pixels);
65                 if(pixels[0] < 10 || pixels[1] < 10 || pixels[2] < 10){
66                     for (i = 0; i < bandas; i++){
67                         rasterEscrita.setSample(coluna, linha, i, 255);
68                     }
69                 }
70                 else {
71                     for (i = 0; i < bandas; i++) {
72                         rasterEscrita.setSample(coluna, linha, i, 0);
73                     }
74                 }
75             }
76         }
77     }

```

```
78     contadorNome++;
79     arquivo = salvaImagens + contadorNome + ".jpg";
80     ImageIO.write(imagem, "jpg", new File(arquivo));
81 } catch (Exception e){
82     System.out.println("Houve erro no processamento da imagem!");
83 }
84 }
85 }
86 public class CarregaImagens {
87     public File [] metodoCarregaImagens (String path) throws IOException {
88         File caminho = new File (path);
89         File [] imagens;
90         FilenameFilter filtro = new FilenameFilter() {
91             @Override
92             public boolean accept(File caminho, String name) {
93                 return !name.endsWith(".db");
94             }
95         };
96         imagens = caminho.listFiles(filtro);
97         return imagens;
98     }
99 }
```

**Algoritmo 9:** Código-fonte do *SoftBorda*.

# *APÊNDICE B*

## *Código-Fonte do SoftSkeleton*

```

1 package esqueletizacaoAutomatica; //SoftSkeleton
2 import java.awt.image.BufferedImage;
3 import java.awt.image.Raster;
4 import java.awt.image.WritableRaster;
5 import java.io.File;
6 import java.io.FilenameFilter;
7 import java.io.IOException;
8 import javax.imageio.ImageIO;
9 public class EsqueletizacaoAutomatica {
10     public static void main(String[] args) throws InterruptedException, IOException {
11         EsqueletizacaoAutomatica esqueletizacao = new EsqueletizacaoAutomatica();
12         esqueletizacao.metodoEsqueletizar();
13     }
14     public void metodoEsqueletizar() throws IOException, InterruptedException {
15         int Contador, largura, altura, linha, coluna, bandas, bands, qtdImagens;
16         int i, j, p2, p3, p4, p5, p6, p7, p8, p9;
17         int escalaCinza, verificaCinza, maiorCinza = 1;
18         int[] vCor;
19         int branco = 255;
20         boolean apagaPixel;
21         String salvarArquivo;
22         File[] imagens;
23         imagens = this.metodoCarregaImagens("C:\\\\ImagensCinza");
24         qtdImagens = imagens.length;

```

```

25     for (j = 0; j < qtdImagens; j++) {
26         try {
27             BufferedImage imagem = ImageIO.read(imagens[j]);
28             largura = imagem.getWidth();
29             altura = imagem.getHeight();
30             Raster rasterLeitura = imagem.getRaster();
31             WritableRaster rasterEscrita = imagem.getRaster();
32             bandas = rasterLeitura.getNumBands();
33             vCor = new int[bandas];
34             do {
35                 apagaPixel = false;
36                 for (linha = 1; linha < altura - 1; linha++) { //Primeiro loop
37                     for (coluna = 1; coluna < largura - 1; coluna++) {
38                         rasterEscrita.getPixel(coluna, linha, vCor);
39                         if (vCor[0] == branco) continue;
40                         escalaCinza = vCor[0];
41                         verificaCinza = 0;
42                         for (i = 0; i < bandas; i++)
43                             verificaCinza += vCor[i];
44                         if (verificaCinza != (vCor[0] * bandas))
45                             continue;
46                         p2 = rasterEscrita.getSample(coluna, linha - 1, 0);
47                         p3 = rasterEscrita.getSample(coluna + 1, linha - 1, 0);
48                         p4 = rasterEscrita.getSample(coluna + 1, linha, 0);
49                         p5 = rasterEscrita.getSample(coluna + 1, linha + 1, 0);
50                         p6 = rasterEscrita.getSample(coluna, linha + 1, 0);
51                         p7 = rasterEscrita.getSample(coluna - 1, linha + 1, 0);
52                         p8 = rasterEscrita.getSample(coluna - 1, linha, 0);
53                         p9 = rasterEscrita.getSample(coluna - 1, linha - 1, 0);
54                         Contador = 0;
55                         if (p2 == escalaCinza) {Contador++;}
56                         if (p3 == escalaCinza) {Contador++;}
57                         if (p4 == escalaCinza) {Contador++;}
58                         if (p5 == escalaCinza) {Contador++;}
59                         if (p6 == escalaCinza) {Contador++;}
60                         if (p7 == escalaCinza) {Contador++;}
61                         if (p8 == escalaCinza) {Contador++;}
62                         if (p9 == escalaCinza) {Contador++;}
63                         if (Contador <= 2 || Contador >= 6) Erro semeado SUT2: "Contador > 6".
64                         {continue;}
65                         Contador = 0;
66                         if (p2 == branco && p3 == escalaCinza) {Contador++;}
67                         if (p3 == branco && p4 == escalaCinza) {Contador++;}
68                         if (p4 == branco && p5 == escalaCinza) {Contador++;}
69                         if (p5 == branco && p6 == escalaCinza) {Contador++;}
70                         if (p6 == branco && p7 == escalaCinza) {Contador++;}
71                         if (p7 == branco && p8 == escalaCinza) {Contador++;}
72                         if (p8 == branco && p9 == escalaCinza) {Contador++;}
73                         if (p9 == branco && p2 == escalaCinza) {Contador++;}
74                         if (Contador != 1) {continue;}
75                         if (p2 == escalaCinza && p4 == escalaCinza && p8 == escalaCinza)
76                         {continue;}
77                         if (p2 == escalaCinza && p6 == escalaCinza && p8 == escalaCinza)
78                         {continue;}
79                         apagaPixel = true;
80                         rasterEscrita.setSample(coluna, linha, 0, maiorCinza);
81                     }
82                 }

```

```

81         for (linha = 0; linha < altura; linha++) {
82             for (coluna = 0; coluna < largura; coluna++) {
83                 if (rasterEscrita.getSample(coluna, linha, 0) == maiorCinza) {
84                     for (bands = 0; bands < bandas; bands++) {
85                         rasterEscrita.setSample(coluna, linha, bands, branco);
86                     }
87                 }
88             }
89         }
90         for (linha = 1; linha < altura - 1; linha++) { //Segundo loop
91             for (coluna = 1; coluna < largura - 1; coluna++) {
92                 rasterEscrita.getPixel(coluna, linha, vCor);
93                 if (vCor[0] == branco)
94                     continue;
95                 escalaCinza = vCor[0];
96                 verificaCinza = 0;
97                 for (i = 0; i < bandas; i++)
98                     verificaCinza += vCor[i];
99                 if (verificaCinza != (vCor[0] * bandas))
100                     continue;
101                p2 = rasterEscrita.getSample(coluna, linha - 1, 0);
102                p3 = rasterEscrita.getSample(coluna + 1, linha - 1, 0);
103                p4 = rasterEscrita.getSample(coluna + 1, linha, 0);
104                p5 = rasterEscrita.getSample(coluna + 1, linha + 1, 0);
105                p6 = rasterEscrita.getSample(coluna, linha + 1, 0);
106                p7 = rasterEscrita.getSample(coluna - 1, linha + 1, 0);
107                p8 = rasterEscrita.getSample(coluna - 1, linha, 0);
108                p9 = rasterEscrita.getSample(coluna - 1, linha - 1, 0);
109                Contador = 0;
110                if (p2 == escalaCinza) {Contador++;}
111                if (p3 == escalaCinza) {Contador++;}
112                if (p4 == escalaCinza) {Contador++;}
113                if (p5 == escalaCinza) {Contador++;}
114                if (p6 == escalaCinza) {Contador++;}
115                if (p7 == escalaCinza) {Contador++;}
116                if (p8 == escalaCinza) {Contador++;}
117                if (p9 == escalaCinza) {Contador++;}
118                if (Contador <= 2 || Contador >= 6)
119                    {continue;}
120                Contador = 0;
121                if (p2 == branco && p3 == escalaCinza) {Contador++;}
122                if (p3 == branco && p4 == escalaCinza) {Contador++;}
123                if (p4 == branco && p5 == escalaCinza) {Contador++;}
124                if (p5 == branco && p6 == escalaCinza) {Contador++;}
125                if (p6 == branco && p7 == escalaCinza) {Contador++;}
126                if (p7 == branco && p8 == escalaCinza) {Contador++;}
127                if (p8 == branco && p9 == escalaCinza) {Contador++;}
128                if (p9 == branco && p2 == escalaCinza) {Contador++;}
129                if (Contador != 1)
130                    {continue;}
131                if (p2 == escalaCinza && p4 == escalaCinza && p6 == escalaCinza)
132                    {continue;}
133                if (p4 == escalaCinza && p6 == escalaCinza && p8 == escalaCinza)
134                    {continue;} // -Erro semeado SUT3: “== escalaCinza || p6”.
135                apagaPixel = true;
136                rasterEscrita.setSample(coluna, linha, 0, maiorCinza);
137            }
138        }

```

```

107         for (linha = 0; linha < altura; linha++) {
108             for (coluna = 0; coluna < largura; coluna++) {
109                 if (rasterEscrita.getSample(coluna, linha, 0) == maiorCinza) {
110                     for (bands = 0; bands < bandas; bands++) {
111                         rasterEscrita.setSample(coluna, linha, bands, branco);
112                     }
113                 }
114             }
115         }
116     } while (apagaPixel == true);
117     salvarArquivo = “C:\\\\Esqueletizacao” + imagens[j].getName();
118     ImageIO.write(imagem, “jpg”, new File(salvarArquivo));
119 } catch (Exception e) {
120     System.out.println(“Excecao: ” + imagens[j].getName());
121     continue;
122 }
123 }
124 System.out.println(“Fim da esqueletização!”);
125 }
126 public File[] metodoCarregaImagens(String path) throws IOException {
127     File caminho = new File(path);
128     File[] imagens;
129     FilenameFilter filtro = new FilenameFilter() {
130         @Override
131         public boolean accept(File caminho, String name) {
132             return !name.endsWith(“.db”);
133         }
134     };
135     imagens = caminho.listFiles(filtro);
136     return imagens;
137 }
138 }
```

**Algoritmo 10:** Código-fonte do *SoftSkeleton*

# *APÊNDICE C*

## *Código-Fonte dos Pacotes, Classes e Funções Implementados*

```

1 package funcoesSimilaridade;
2 public class Euclidiana {
3     public double distancia (double[] vectModel, double[] vectTesting) {
4         double d1, d2, distance = 0.0;
5         for (int i = 0; i < vectModel.length; i++) {
6             d1 = vectModel[i];
7             d2 = vectTesting[i];
8             distance += (d1 - d2) * (d1 - d2);
9         return Math.sqrt(distance); }
10    public class Canberra {
11        public double distancia (double[] vectModel, double[] vectTesting) {
12            double d1, d2, distance = 0.0;
13            for(int i = 0; i < vectModel.length; i++) {
14                d1 = vectModel[i];
15                d2 = vectTesting[i];
16                if(d1 == 0.0 && d2 == 0.0) {
17                    distance += 0.0;
18                } else {
19                    distance += ( Math.abs(d1-d2) / (Math.abs(d1) + Math.abs(d2)));
20                }
21            return distance; }
22            public class CarregaFuncaoSimilaridade {
23                public Object carregaFuncaoSimilaridade(String funcaoSimilaridade, double vectModel [], double vectTesting []) {
24                    Object funcSimilaridade = null;
25                    switch (funcaoSimilaridade) {
26                        case "Canberra":
27                            funcSimilaridade = new Canberra().distancia(vectModel, vectTesting);
28                            break;
29                        case "Euclidiana":
30                            funcSimilaridade = new Euclidiana().distancia(vectModel, vectTesting);
31                            break; }
32            return funcSimilaridade; } }
```

**Algoritmo 11:** Código-fonte das funções de similaridade.

```
1 package estatisticas;
2 import java.io.IOException;
3 import java.math.BigDecimal;
4 import java.util.*;
5 public class Estatisticas {
6     private double array[];
7     public double getSomaDosElementos() {
8         double total = 0;
9         for (int counter = 0; counter < array.length; counter++)
10            total += array[counter];
11        return total;
12    }
13    public double getSomaDosElementosAoQuadrado() {
14        double total = 0;
15        for (int counter = 0; counter < array.length; counter++)
16            total += Math.pow(array[counter], 2);
17        return total;
18    }
19    public double getVariancia() {
20        double p1 = 1 / Double.valueOf(array.length - 1);
21        double p2 = getSomaDosElementosAoQuadrado() - (Math.pow(getSomaDosElementos(), 2) /
22            Double.valueOf(array.length));
23        return p1 * p2;
24    }
25    public double getMedia() {
26        return getSomaDosElementos() / array.length;
27    }
28    public double getVariancia2() {
29        double total = 0;
30        double media = getMedia();
31        for (int counter = 0; counter < array.length; counter++) {
32            double d = array[counter] - media;
33            total += d * d;
34        }
35        return total / (double) (array.length - 1);
36    }
37    public double getDesvioPadrao() {
38        return Math.sqrt(getVariancia());
39    }
40    public double getDesvioPadrao2() {
41        return Math.sqrt(getVariancia());
42    }
43    public void setArray(double[] array) {
44        this.array = array;
45    }
46    public double getMaiorRaio() {
47        double maior = array[0];
48        for (int i = 1; i < array.length; i++) {
49            if (array[i] > maior) maior = array[i];
50        }
51        return maior;
52    }
```

```

1 public double [] getEstatisticas (double [][] matrizSimilaridade) throws IOException{
2 int linha, coluna, tamanho, posicaoQuartil1, posicaoQuartil2, posicaoQuartil3, maiorColuna,
tamanhoColuna;
3 ArrayList listaSimilaridade = new ArrayList();
4 double [] estatisticas = new double[6];
5 double total = 0, media, menorValor, maiorValor;
6 maiorColuna = 0;
7 for (linha = 0; linha < matrizSimilaridade.length; linha++) {
8 tamanhoColuna = matrizSimilaridade[linha].length;
9 if(tamanhoColuna > maiorColuna){
10 maiorColuna = tamanhoColuna;
11 }
12 }
13 for (coluna = 0; coluna < matrizSimilaridade.length; coluna++) {
14 for (linha = 0; linha < matrizSimilaridade.length; linha++) {
15 try {
16 if (matrizSimilaridade[linha][coluna] != 0)
17 listaSimilaridade.add(matrizSimilaridade[linha][coluna]);
18 }
19 catch (Exception e) { }
20 }
21 }
22 Collections.sort(listaSimilaridade);
23 tamanho = listaSimilaridade.size();
24 for (linha = 0; linha < tamanho; linha++){
25 total += (double) listaSimilaridade.get(linha);
26 }
27 media = new
    BigDecimal(total/tamanho).setScale(6,BigDecimal.ROUND_HALF_UP).doubleValue();
28 menorValor = new BigDecimal((double) listaSimilaridade.get(0)) .setScale(6,
    BigDecimal.ROUND_HALF_UP).doubleValue();
29 maiorValor = new BigDecimal((double)listaSimilaridade.get(tamanho-1)) .setScale(6,
    BigDecimal.ROUND_HALF_UP).doubleValue();
30 posicaoQuartil1 = (int) Math.round(0.25*(tamanho+1));
31 posicaoQuartil2 = (int) Math.round(0.5*(tamanho+1));
32 posicaoQuartil3 = (int) Math.round(0.75*(tamanho+1));
33 estatisticas [0] = menorValor;
34 estatisticas [1] = maiorValor;
35 estatisticas [2] = media;
36 estatisticas [3] = new BigDecimal((double)listaSimilaridade.get(posicaoQuartil1)).setScale(6,
    BigDecimal.ROUND_HALF_UP).doubleValue();
37 if (tamanho % 2 == 0)
38 estatisticas [4] = new BigDecimal((double)listaSimilaridade.get(tamanho/2) +
    (double)listaSimilaridade.get((tamanho/2)+1)/2).setScale(6,BigDecimal.ROUND_HALF_UP
    .doubleValue());
39 else
40 estatisticas [4] = new BigDecimal((double) listaSimilaridade.get(posicaoQuartil2)).setScale(6,
    BigDecimal.ROUND_HALF_UP).doubleValue();
41 estatisticas [5] = new BigDecimal((double)listaSimilaridade.get(posicaoQuartil3)) .setScale(6,
    BigDecimal.ROUND_HALF_UP).doubleValue();
42 return estatisticas;
43 }
44 }
```

**Algoritmo 12:** Código-fonte das funções estatísticas.

```
1 package processamentoImagens;
2 import extractores.*;
3 import java.io.File;
4 import java.io.FilenameFilter;
5 import java.io.IOException;
6 public class CarregaImagens {
7     public File [] metodoCarregaImagens (String path) throws IOException {
8         File caminho = new File (path);
9         File [] imagens;
10        FilenameFilter filtro = new FilenameFilter() {
11            @Override
12            public boolean accept(File caminho, String name) {
13                return !name.endsWith(".db");
14            }
15        };
16        imagens = caminho.listFiles(filtro);
17        return imagens;
18    }
19}
20 public class CriaMatrizCaracteristicas {
21    public double [][] metodoMatrizCaracteristicas (String path) throws IOException {
22        CarregaImagens getImagens = new CarregaImagens();
23        File [] imagens = getImagens.metodoCarregaImagens(path);
24        int contadorLinha, contadorColuna = 0;
25        int qtdImagens = imagens.length;
26        int qtdExtratores = 3;
27        double [][] matrizCaracteristicas = new double [qtdImagens][qtdExtratores];
28        ExtratorArea area = new ExtratorArea ();
29        ExtratorPerimetro perimetro = new ExtratorPerimetro();
30        ExtratorCor cor = new ExtratorCor();
31        //Calcula a área
32        for (contadorLinha=0; contadorLinha < qtdImagens; contadorLinha++){
33            matrizCaracteristicas [contadorLinha][contadorColuna] =
34                area.areaValue(imagens[contadorLinha]);
35        }
36        contadorColuna++;
37        //Calcula o perímetro
38        for (contadorLinha=0; contadorLinha < qtdImagens; contadorLinha++){
39            matrizCaracteristicas [contadorLinha][contadorColuna] =
40                perimetro.perimetroValue(imagens[contadorLinha]);
41        }
42        contadorColuna++;
43        //Calcula a média de cores
44        for (contadorLinha=0; contadorLinha < qtdImagens; contadorLinha++){
45            matrizCaracteristicas [contadorLinha][contadorColuna] = cor.corValue(imagens[contadorLinha]);
46        }
47    }
48    return matrizCaracteristicas;
49}
```

```

1 public File [] metodoListaArquivos(String path){
2     File caminho = new File (path); //path
3     File [] listaArquivos;
4     FilenameFilter filtro = new FilenameFilter(){
5         @Override
6         public boolean accept(File caminho, String name){
7             return !name.endsWith(".db");
8         }
9     };
10    listaArquivos = caminho.listFiles(filtro);
11    return listaArquivos;
12 }
13 }
```

**Algoritmo 13:** Código-fonte das funções de processamento de imagens.

```

1 package clustering;
2 import funcoesSimilaridade.*;
3 import java.io.IOException;
4 import estatisticas.*;
5 import java.io.File;
6 import java.io.IOException;
7 import java.util.Arrays;
8 import processamentoImagens.*;
9 public class CriaMatrizSimilaridade {
10    public double[][] metodoMatrizSimilaridade(String path, String funcaoSimilaridade) throws
11        IOException {
12        double matrizSimilaridade[][];
13        double matrizCaracteristicas[][];
14        int qtdimagens, linha, coluna;
15        CriaMatrizCaracteristicas caracteristicas = new CriaMatrizCaracteristicas();
16        matrizCaracteristicas = caracteristicas.metodoMatrizCaracteristicas(path);
17        qtdimagens = matrizCaracteristicas.length;
18        matrizSimilaridade = new double[qtdimagens - 1][];
19        CarregaFuncaoSimilaridade funcSimilaridade = new CarregaFuncaoSimilaridade();
20        for (linha = 1; linha < qtdimagens; linha++) { matrizSimilaridade[linha - 1] = new double[linha];
21        for (coluna = 0; coluna < matrizSimilaridade.length; coluna++) {
22            if (linha == coluna) {
23                break;
24            }
25            matrizSimilaridade[linha - 1][coluna] = (double)
26            funcSimilaridade.carregaFuncaoSimilaridade(funcaoSimilaridade, matrizCaracteristicas[linha],
27            matrizCaracteristicas[coluna]);
28        }
29    }
30    return matrizSimilaridade;
31 }
```

```

1 public class ClusteringHierarquico {
2     public File[][] ligacaoSimples(String path, String funcaoSimilaridade, double pararClustering)
3         throws IOException {
4         double matrizSimilaridade[][];
5         double vetorEstatisticas[];
6         double menorMatriz = 0;
7         int clusterNovo = 1, i, linha, coluna, qtdimagens, qtdClusters, contadorVetor, contadorLinha,
8             organizaCluster;
9         int colunaAux = 0, linhaAux = 0, menorPosicao, maiorPosicao, maiorVetor = 0, contador,
10            posicaoMenor, posicaoMaior;
11        CriaMatrizSimilaridade similaridade = new CriaMatrizSimilaridade();
12        matrizSimilaridade = similaridade.metodoMatrizSimilaridade(path, funcaoSimilaridade);
13        Estatisticas estatisticas = new Estatisticas();
14        vetorEstatisticas = estatisticas.getEstatisticas(matrizSimilaridade);
15        pararClustering = vetorEstatisticas[3];
16        File[][] vetorImagens[];
17        CarregaImagens imagens = new CarregaImagens();
18        vetorImagens = imagens.metodoCarregaImagens(path);
19        qtdimagens = vetorImagens.length;
20        int[] vetCluster = new int[qtdimagens];
21        int[] vetClusterOrdenado = new int[qtdimagens];
22        File[][] matrizClustering;
23        //Início do processo de clustering
24        //Encontra o menor valor da matriz de similaridade e sua respectiva posicao
25        while (menorMatriz < pararClustering) {
26            menorMatriz = Integer.MAX_VALUE;
27            for (coluna = 0; coluna < qtdimagens; coluna++) {
28                for (linha = 0; linha < qtdimagens; linha++) {
29                    try {
30                        if (matrizSimilaridade[linha][coluna] != 0 && matrizSimilaridade[linha][coluna] < menorMatriz) {
31                            menorMatriz = matrizSimilaridade[linha][coluna];
32                            colunaAux = coluna;
33                            linhaAux = linha;
34                        } } catch (Exception e) { } }
35            //Clustering hierarquico
36            if (menorMatriz > pararClustering) {
37                break; }
38            if (colunaAux < linhaAux) {
39                posicaoMenor = colunaAux;
40                posicaoMaior = linhaAux;
41            } else {
42                posicaoMenor = linhaAux;
43                posicaoMaior = colunaAux;
44            }
45            for (coluna = 0; coluna < qtdimagens; coluna++) {
46                if (coluna == posicaoMenor) {
47                    for (i = posicaoMenor; i < matrizSimilaridade[posicaoMaior].length; i++) {
48                        matrizSimilaridade[posicaoMaior][i] = 0;
49                    }
50                }
51                if (matrizSimilaridade[colunaAux][coluna] < matrizSimilaridade[linhaAux][coluna]) {
52                    matrizSimilaridade[linhaAux][coluna] = 0;
53                }
54            }
55        }
56        //Fim do trecho Atualiza os valores da matriz

```

```
1 If (vetCluster[linhaAux] == 0 && vetCluster[colunaAux] == 0) {
2   vetCluster[linhaAux] = clusterNovo;
3   vetCluster[colunaAux] = clusterNovo;
4   clusterNovo++;
5 } else if (vetCluster[linhaAux] == 0) {
6   vetCluster[linhaAux] = vetCluster[colunaAux];
7 } else if (vetCluster[colunaAux] == 0) {
8   vetCluster[colunaAux] = vetCluster[linhaAux];
9 } else if (vetCluster[linhaAux] < vetCluster[colunaAux]) {
10  menorPosicao = vetCluster[linhaAux];
11  maiorPosicao = vetCluster[colunaAux];
12  for (i = 0; i < vetCluster.length; i++) {
13    if (vetCluster[i] == maiorPosicao) {
14      vetCluster[i] = menorPosicao; } }
15 } else if (vetCluster[linhaAux] > vetCluster[colunaAux]) {
16  menorPosicao = vetCluster[colunaAux];
17  maiorPosicao = vetCluster[linhaAux];
18  for (i = 0; i < vetCluster.length; i++) {
19    if (vetCluster[i] == maiorPosicao) {
20      vetCluster[i] = menorPosicao; } }
21 } /* Fim do while*/
22 //Fim do clustering hierarquico
23 //Cria a matriz de clusters
24 for (i = 0; i < vetCluster.length; i++) {
25  if (vetCluster[i] > maiorVetor) {
26    maiorVetor = vetCluster[i]; } }
27 for (i = 0; i < vetCluster.length; i++) {
28  vetClusterOrdenado[i] = vetCluster[i]; }
29 Arrays.sort(vetClusterOrdenado);
30 organizaCluster = 0;
31 qtdClusters = 1;
32 for (i = 0; i < vetCluster.length; i++) {
33  if (vetClusterOrdenado[i] != organizaCluster) {
34    organizaCluster = vetClusterOrdenado[i];
35    qtdClusters++; } }
36 matrizClustering = new File[qtdClusters][];
37 contadorLinha = 0;
38 contador = 0;
39 contadorVetor = 0;
```

```
1 while (contadorVetor < maiorVetor + 1) {
2     for (coluna = 0; coluna < vetCluster.length; coluna++) {
3         if (vetCluster[coluna] == contadorVetor) {
4             contador++;
5         }
6         if (contador > 0) {
7             matrizClustering[contadorLinha] = new File[contador];
8             contadorLinha++;
9         }
10        contador = 0;
11        contadorVetor++;
12    }
13        contadorVetor = 0;
14        contador = 0;
15        linha = 0;
16        contadorLinha = 0;
17        while (contadorVetor < maiorVetor + 1) {
18            for (coluna = 0; coluna < vetCluster.length; coluna++) {
19                if (vetCluster[coluna] == contadorVetor) {
20                    matrizClustering[contadorLinha][contador] = vetorImagens[coluna];
21                    contador++;
22                    linha++;
23                }
24            }
25            if (contador > 0) {
26                contadorLinha++;
27                contador = 0;
28            }
29            contadorVetor++;
30        }
31        return matrizClustering;
32    }
33}
```

**Algoritmo 14:** Código-fonte das funções de *clustering*.

## ***APÊNDICE D***

### ***Métodos Estatísticos***

Este apêndice apresenta os principais conceitos estatísticos empregados neste trabalho e tem como base os estudos de Devore [47] e DeGroot [48].

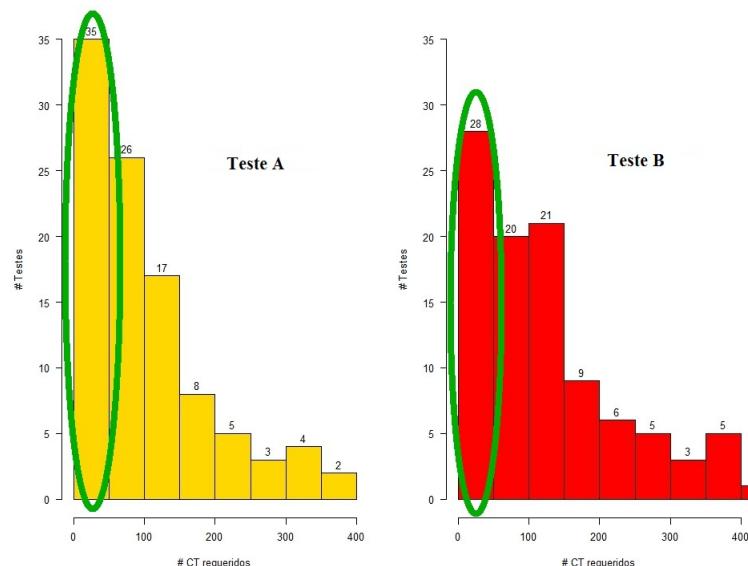
Em pesquisas empíricas, estudar o conjunto de todos os elementos ou resultados sob investigação (conhecido como população) pode acarretar em um enorme gasto de tempo e de recursos, tornando o trabalho inviável. Para mitigar esse problema é possível selecionar uma parte representativa da população sob estudo, chamada de amostra, de forma que as conclusões do estudo dessa amostra possam ser estendidas para a população inteira. Entretanto, quando se trabalha com amostras, surgem algumas possibilidades incertas (mas previsíveis) que devem ser consideradas, como por exemplo: 1) qual o tamanho adequado da amostra? 2) quais os fatores que influenciam os resultados? 3) qual método de avaliação utilizar? Tais possibilidades, quando não tratadas de forma correta e sistemática, podem ocasionar em resultados incompletos, dúbios e até mesmo equivocados. Para diminuir os riscos associados e aumentar a confiabilidade dos resultados, pode-se utilizar a estatística.

A estatística que é uma ciência que tem por objetivo coletar, analisar e interpretar dados numéricos a respeito de fenômenos coletivos ou de massa, bem como induzir as leis a que tais fenômenos obedecem. Cabe à estatística a representação numérica e comparativa, em tabelas ou gráficos, dos resultados da análise dos fenômenos observados.

Com o objetivo de apresentar os principais conceitos estatísticos necessários para a execução do trabalho proposto, esta seção engloba alguns métodos estatísticos amplamente utilizado em estudos empíricos, tais como cálculos de variabilidade, histograma, *boxplot*, *scatterplot*, teste de correlação, teste de qui-quadrado e cálculo do *p-valor*.

A *variabilidade* dos dados é a dispersão de valores de uma variável em torno de um valor tomado como ponto de comparação, como a média ( $\mu$ ), a mediana, o desvio-padrão ( $\sigma$ ), dentre outros. Quanto menor a variabilidade dos dados, maior a sua consistência.

A distribuição de frequências é o agrupamento de dados em classes, de tal forma que se pode quantificar as ocorrências<sup>§</sup> em cada classe. O número de ocorrências de uma determinada classe recebe o nome de frequência absoluta. O *histograma* é um gráfico utilizado para representar a distribuição de frequências de um conjunto de dados quantitativos. No exemplo apresentado, ilustrado na Figura 38, o *eixo X* dos histogramas (que representa as classes) mostra a quantidade de casos de teste requeridos para revelar a presença dos erros de *software*, e o *eixo Y* representa a quantidade de testes de *software* executados. Uma maior distribuição (concentração) dos dados à esquerda do gráfico indica que, em geral, foram necessários menos casos de teste para revelar os erros.



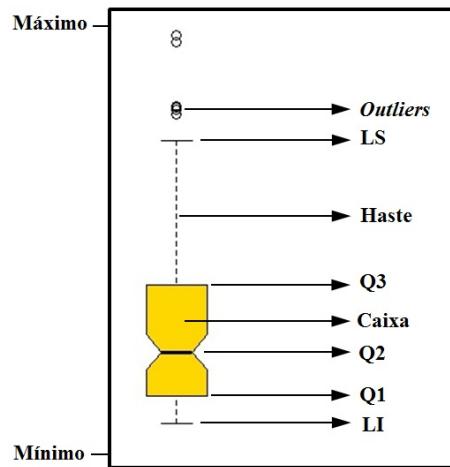
**Figura 38** – Exemplo de um histograma utilizado em análises estatísticas

A Figura 38 ilustra um histograma que mostra o comportamento dos testes de *software*. Nas colunas destacadas pelas elipses, o histograma *Teste A* possui maior densidade à esquerda em relação ao histograma *Teste B*. Isso significa que o *Teste A* apresenta melhores resultados, pois o histograma mostra que em 35 testes os resultados obtidos encontram-se entre as classes “0-50 casos de teste requeridos para revelar a presença do erro semeado” e o *Teste B* mostra que em 28 testes os resultados obtidos encontram-se entre as classes “0-50 casos de teste requeridos para revelar a presença do erro semeado”.

O *boxplot* é um gráfico que permite analisar a distribuição dos dados. Ele apresenta

<sup>§</sup>A quantidade de ocorrências também é conhecida como frequência ou densidade dos dados.

os valores “mínimo, limite inferior ( $LI$ ), primeiro quartil ( $Q1$ ), segunda quartil (mediana,  $Q2$ ), terceiro quartil ( $Q3$ ), limite superior ( $LS$ ) e máximo” do conjunto de dados. A Figura 39 ilustra o *boxplot*.



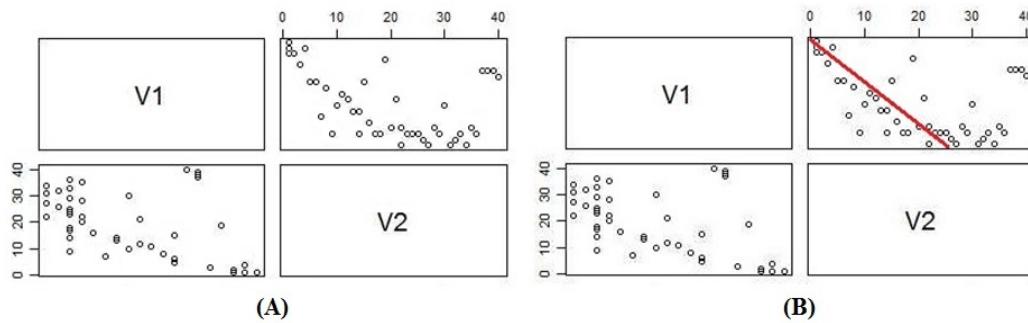
**Figura 39 – Ilustração do gráfico boxplot**

A Figura 39 mostra a estrutura do *boxplot*. O gráfico é composto por uma caixa que representa a parte central (50%) dos dados, posicionada paralelamente ao eixo da escala dos dados. Os limites inferiores e superiores são obtidos por meio do seguinte cálculo:  $LI = Q1 - 1,5(Q3 - Q1)$  e  $LS = Q3 + 1,5(Q3 - Q1)$ . A haste inferior se estende desde  $LI$  até  $Q1$  e a haste superior se estende desde o  $Q3$  até  $LS$ . A linha interna representa a mediana ( $Q2$ ); quanto maior a caixa, mais dispersos são os dados. Em caso de *boxplots* comparativos, é possível observar como as caixas se comportam e o deslocamento entre as caixas.

No *boxplot*, quanto maior a simetria dos dados, menor a variabilidade. A simetria pode ser verificada por meio da análise da posição dos pontos  $Q1$ ,  $LI$ ,  $Q2$ ,  $Q3$  e  $LS$ . Comprimentos muito diferentes entre os pontos caracterizam assimetria. Os dados que ultrapassam os limites inferior ou superior são considerados atípicos (*outliers*).

O *scatterplot* é um tipo de gráfico que mostra os valores para duas variáveis de um conjunto de dados. Os dados são apresentados como um conjunto de pontos, no qual o valor de uma variável  $X$  determina a posição sobre o eixo horizontal, e o valor da outra variável  $Y$  determina a posição sobre o eixo vertical.

É possível acrescentar ao *scatterplot* uma linha de regressão linear para avaliar, por meio de simulação, os efeitos sobre uma variável  $Y$  em decorrência de alterações introduzidas nos valores de outra variável  $X$ . A regressão linear considera que a relação da resposta às variáveis é uma função linear. A Figura 40 ilustra o *scatterplot*.



**Figura 40 – Ilustração do gráfico scatterplot**

A Figura 40A mostra a estrutura do *scatterplot* para as variáveis  $V1$  e  $V2$ , e a Figura 40B mostra o *scatterplot* com a linha de regressão. É importante ressaltar que o gráfico sugere um tipo de relacionamento *linear* entre as variáveis  $V1$  e  $V2$ , pois a maioria dos dados segue o comportamento da reta de regressão.

O *scatterplot* mostra o comportamento das variáveis, enquanto o teste de correlação explicita a força da relação linear entre as variáveis. O coeficiente de correlação  $\rho$ , apresentado na equação D.1, varia entre -1 e +1. Quanto mais próximo de zero for o resultado, menor a correlação entre as variáveis.

$$\rho_{X,Y} = \frac{cov_{X,Y}}{\sigma_X \sigma_Y}, \quad \sigma_X = \sqrt{E[X^2] - (E[X])^2} \quad (\text{D.1})$$

$$cov_{X,Y} = E[(X - \mu_X)(Y - \mu_Y)], \quad \mu_X = \frac{1}{n} \sum_{i=1}^n x_i \quad (\text{D.2})$$

$$E[X] = \sum_{i=1}^{\infty} x_i p(x_i) \quad (\text{D.3})$$

A covariância  $cov_{X,Y}$  (equação D.2) é uma medida que expressa a variabilidade conjunta de duas variáveis;  $\sigma_X$  representa o cálculo do desvio-padrão e mostra o quanto de variação ou “dispersão” existe em relação à média; o valor esperado  $E[X]$  (equação D.3) de uma variável é a soma das probabilidades de cada possibilidade de saída da experiência multiplicada pelo seu valor, ou seja, representa o valor médio “esperado” de uma experiência se ela for repetida várias vezes e  $\mu_X$  representa o cálculo da média simples.

O teste de qui-quadrado ( $\chi^2$ ) permite analisar o nível de associação (dependência) entre variáveis. O  $\chi^2$  é um teste de hipóteses que se destina a encontrar o valor da dispersão para duas variáveis. O princípio básico deste método é comparar proporções, isto é, as possíveis divergências entre as frequências observadas e esperadas para um certo

evento. Quanto maior o valor de  $\chi^2$ , mais significante é a relação entre as variáveis. A equação D.4 apresenta o cálculo de  $\chi^2$ .

$$\chi^2 = \sum \frac{(O - E[Y])^2}{E[Y]} \quad (\text{D.4})$$

Na equação D.4,  $(O - E[Y])$  é a diferença entre a frequência observada e a esperada da variável  $Y$  em uma classe. As frequências observadas são obtidas diretamente dos dados amostrais, enquanto que as frequências esperadas são calculadas a partir destas. Quando as frequências observadas são muito próximas às esperadas, o valor de  $\chi^2$  é pequeno. Entretanto, quando as divergências são grandes  $(O - E[Y])$  também é grande e, consequentemente,  $\chi^2$  assume um valor alto.

Em testes estatísticos, a presunção de que não existe associação entre as variáveis é conhecida como *hipótese nula* ( $H_o$ ). O  $\chi^2$  é uma comparação dos valores observados com os valores esperados em caso de hipótese nula.

Para se atribuir um nível de confiabilidade aos testes estatísticos, normalmente calcular-se um nível descritivo conhecido como *p-valor*. Calcular o *p-valor* ajuda a verificar se os resultados obtidos foram alcançados devido ao acaso ou se são frutos dos fatores que estão sendo analisados. A equação D.5 apresenta o cálculo do *p-valor*.

$$p = P[Z \geq z | H_o], \quad z = \frac{x_i - \mu}{\sigma} \quad (\text{D.5})$$

O *p-valor* pode variar entre zero e um, e seu resultado pode ser descrito como:

“Supondo que o resultado do experimento tenha acontecido devido ao acaso ( $H_o$ ), qual seria a probabilidade de se observar um resultado tão extremo ou mais extremo que o da amostra ( $P[Z \geq z]$ )?”.

Em muitas áreas do conhecimento admite-se um valor limite de  $p$  menor ou igual a 0,05, ou seja, assume-se como margem de segurança 5% de chances de erro (ou 95% de chances de acerto). Como o *p-valor* é calculado supondo que  $H_o$  seja verdadeira, pode-se fazer duas avaliações quando se obtém um resultado muito pequeno: 1) um evento que é extremamente raro pode ter ocorrido ou 2) a hipótese  $H_o$  não deve ser verdadeira, isto é, não parece plausível. Quanto menor o *p-valor*, maior a evidência para rejeitar  $H_o$ .