MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/03.29.11.00.21-TDI

UM ALGORITMO PARA GERAÇÃO DE CASOS DE TESTE COMBINATORIAL VIA MATRIZ DE COBERTURA COM NÍVEIS VARIADOS

Juliana Marino Balera

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 14 de fevereiro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3NK7TBE>

INPE
São José dos Campos
2017

PUBLICADO POR:

Instituto Nacional de Pesquisas Espaciais - INPE
Gabinete do Diretor (GB)
Serviço de Informação e Documentação (SID)
Caixa Postal 515 - CEP 12.245-970
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/6921
E-mail: pubtc@inpe.br

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

sid.inpe.br/mtc-m21b/2017/03.29.11.00.21-TDI

UM ALGORITMO PARA GERAÇÃO DE CASOS DE TESTE COMBINATORIAL VIA MATRIZ DE COBERTURA COM NÍVEIS VARIADOS

Juliana Marino Balera

Dissertação de Mestrado do Curso de Pós-Graduação em Computação Aplicada, orientada pelo Dr. Valdivino Alexandre de Santiago Júnior, aprovada em 14 de fevereiro de 2017.

URL do documento original:
<http://urlib.net/8JMKD3MGP3W34P/3NK7TBE>

INPE
São José dos Campos
2017

Dados Internacionais de Catalogação na Publicação (CIP)

Balera, Juliana Marino.

B195a Um algoritmo para geração de casos de teste combinatorial via matriz de cobertura com níveis variados / Juliana Marino Balera. – São José dos Campos : INPE, 2017.

xviii + 89 p. ; (sid.inpe.br/mtc-m21b/2017/03.29.11.00.21-TDI)

Dissertação (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2017.

Orientador : Dr. Valdivino Alexandre de Santiago Júnior.

1. Teste de Software. 2. T-Tuple Reallocation. 3. Designs Combinatoriais. 4. Matriz de Cobertura com Níveis Variados. 5. Experimento Controlado. I.Título.

CDU 004.421

Aluno (a): *Juliana Marino Balera*

Título: "UM ALGORITMO PARA GERAÇÃO DE CASOS DE TESTE COMBINATORIAL VIA MATRIZ DE COBERTURA COM NÍVEIS VARIADOS"

Aprovado (a) pela Banca Examinadora em cumprimento ao requisito exigido para obtenção do Título de *Mestre* em *Computação Aplicada*

Dr. Celso Luiz Mendes

Presidente / INPE / São José dos Campos - SP

Dr. Valdivino Alexandre de Santiago Júnior

Orientador(a) / INPE / São José dos Campos - SP

Dr. Eduardo Martins Guerra

Membro da Banca / INPE / São José dos Campos - SP

Dr. Luiz Eduardo Galvão Martins

Convidado(a) / UNIFESP / São José dos Campos - SP

Este trabalho foi aprovado por:

( ) maioria simples

(X) unanimidade

São José dos Campos, 14 de fevereiro de 2017

AGRADECIMENTOS

Agradeço a minha família, aos meus amigos, ao meu orientador e ao Instituto Nacional de Pesquisas Espaciais (INPE), que tornaram possível esse trabalho. Agradeço também, a Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) que financiou essa pesquisa.

RESUMO

Na perspectiva de sistemas complexos, como softwares desenvolvidos para aplicações espaciais tais como satélites, balões estratosféricos e foguetes, existem sempre riscos relacionados ao mau funcionamento do produto que podem causar danos ao meio ambiente, grandes perdas financeiras, ou o pior, perda de vidas. Para minimizar ao máximo esses riscos, é necessário que o processo de teste desses sistemas ocorra de forma rigorosa e eficiente. Como não é possível testar tais produtos exaustivamente, dada a larga gama de casos de teste possíveis, é fundamental, portanto, que se tenham disponíveis métodos para a geração/seleção de casos de teste que possuem grande potencial de revelação de defeitos, e que possuam custo reduzido. Nessa direção, *designs* combinatoriais vêm chamando atenção da comunidade de teste de software para gerar conjuntos de casos de testes menores (menor custo para executar) e eficientes (capacidade de encontrar defeitos no software). Diante disso, essa dissertação de mestrado tem como objetivo apresentar uma nova forma de gerar conjuntos de casos de teste via *designs* combinatoriais, sendo que tais casos de teste tenham custo menor e eficiência comparável à soluções já existentes na literatura. Então, um algoritmo, denominado *T-Tuple Reallocation* (TTR; Realocação de T-Tuplas), para gerar casos de teste de software via *designs* combinatoriais, especificamente via a técnica de Matriz de Cobertura com Níveis Variados (MCNV), foi desenvolvido. A ideia geral do TTR é derivar uma MCNV M por meio da criação e realocação de *t-tuplas* para a matriz M, considerando um parâmetro chamado *meta* (ζ). Dois experimentos controlados e um quasiexperimento foram realizados para comparar o TTR com outros quatro algoritmos/ferramentas bastante conhecidos que geram MCNVs. No primeiro experimento controlado, comparou-se duas perspectivas de custo considerando a versão 1.1 do algoritmo TTR: tamanho das *suites* de teste e tempo para gerar as *suites* de teste. Além disso, realizou-se uma análise de similaridade entre esses conjuntos. No segundo experimento controlado, foi considerada uma versão melhorada do algoritmo TTR, versão 1.2, e comparou-se com os mesmos quatro algoritmos/ferramentas anteriores, mas considerando somente a perspectiva de custo relacionada ao tamanho das *suites* de teste e análise de similaridade. Por fim, um quasiexperimento foi realizado onde comparou-se a eficiência entre o TTR 1.2 e as outras quatro soluções, usando análise de mutantes e aplicando a um estudo de caso da área espacial. As conclusões dessas três avaliações rigorosas são que o TTR foi o algoritmo que apresentou melhor custo (menor quantidade de casos de teste para serem executados), mas que não há diferença de eficiência entre o TTR e as demais soluções. Além disso, as *suites* de teste não são similares, comparando o TTR com as outras soluções. Desse modo, pode-se afirmar que o TTR foi superior aos demais algoritmos/ferramentas pois teve mesma eficiência mas melhor custo.

Palavras-chave: Teste de Software. *T-Tuple Reallocation*. *Designs* Combinatoriais. Matriz de Cobertura com Níveis Variados. Experimento Controlado. Quasiexperimento.

An Algorithm for Generating Combinatorial Test Cases via Mixed-Level Covering Array

With respect to complex systems, such as software developed for space applications like satellites, stratospheric balloons and rockets, there are always risks related to product malfunctioning that can cause damage to the environment, great financial losses, or worse, loss of lives. To minimize these risks, the testing process of these systems must be rigorous and efficient. Since it is not possible to test such products exhaustively, given the wide range of possible test cases, it is critical, therefore, that there are available methods for the generation/selection of test cases which have great potential for defects detection and a reduced cost. In this direction, combinatorial designs have drawn attention of the software testing community to generate sets of smaller (lower cost to run) and efficient (ability to find software defects) test cases. Therefore, this master dissertation aims to present a new way for generating sets of test cases via combinatorial designs, where such test cases have a lower cost and efficiency comparable to solutions already existing in the literature. Then, an algorithm, called T-Tuple Reallocation (TTR), to generate software test cases via combinatorial designs, specifically via the Mixed-Level Covering Array technique (MCA) was developed. The main reasoning behind TTR is to derive an MCA M by creating and relocating t-tuples to the matrix M, considering a parameter called goal (ζ). Two controlled experiments and one quasiexperiment were performed to compare TTR with four other well-known algorithms/tools that generate MCAs. In the first controlled experiment, version 1.1 of TTR was compared considering two cost perspectives: size of the test suites and time to generate the test suites. In addition, a similarity analysis was accomplished between these sets. In the second controlled experiment, an improved version of TTR, version 1.2, was compared with the same four previous algorithms/tools, but only in the cost perspective related to the size of the test suites and similarity analysis. Finally, a quasiexperiment aimed to assess the efficiency between TTR 1.2 and the other four solutions was carried out, via mutation testing and a space application case study. The conclusions of these three rigorous evaluations are that TTR was the algorithm that presented the better cost (smaller number of test cases to execute), but that there is no difference in efficiency between TTR and the other solutions. In addition, the test suites are not similar, comparing TTR with the other approaches. Thus, it can be asserted that the TTR algorithm was superior to the other algorithms/tools because it had the same efficiency but better cost.

Keywords: Software Testing. T-Tuple Reallocation. Combinatorial Designs. Mixed-Level Covering Array. Controlled Experiment. Quasiexperiment.

LISTA DE FIGURAS

Pág.

LISTA DE TABELAS

Pág.

LISTA DE ABREVIATURAS E SIGLAS

ACTS	–	*Advanced Combinatorial test System*
AETG	–	*Automatic Efficient Test Generator*
AG	–	Algoritmo Genético
BT	–	Busca Tabu
CLI	–	*Command Line Interface*
CUDA	–	*Compute Unified Device Architecture*
ED	–	*Experimental Designs*
GPU	–	*Graphics Processing Unit*
INPE	–	Instituto Nacional de Pesquisas Espaciais
IONEX	–	*Ionospheric Plasma Experiments*
IPO	–	*In-Parameter-Order*
IPOG	–	*In-Parameter-Order General*
MCNV	–	Matriz de Cobertura com Níveis Variados
MC	–	Matriz de Cobertura
MO	–	Matriz Ortogonal
MONV	–	Matriz Ortogonal com Níveis Variados
OBDH	–	*On-Board Data Handling*
POO	–	Programação Orientada a Objetos
PDC	–	*Payload Data Handling Computer*
PICT	–	*Pairwise Indepedent Combinatorial Test*
QSEE	–	*Quality of Space Application Embedded Software*
RBM	–	*Recursive Block Method*
RS	–	*Event Pre-Processors*
SA	–	*Simulated Annealing*
SWPDC	–	*Software Payload Data Handling Computer*
TSA	–	*Tabu Search Approach*
TTR	–	*T-Tuple Reallocation*
UML	–	*Unified Markup Language*
USB	–	*Universal Serial Bus*

SUMÁRIO

Pág.

1 INTRODUÇÃO

Considerando sistemas complexos, tais como softwares desenvolvidos para aplicações espaciais como satélite, balões estratosféricos e foguetes, onde defeitos no software podem ocasionar desastres de grandes proporções, perdas financeiras ou mesmo danos ao meio ambiente, se torna necessário que o controle de qualidade seja feito de maneira rigorosa. Por esse motivo, é muito importante que se tenham disponíveis métodos/técnicas de Teste de Software com grande potencial para a revelação de defeitos, uma vez que realizar teste de maneira exaustiva é impraticável (SANTIAGO JÚNIOR et al., 2008).

Uma das atividades do processo de Teste de Software mais estudadas é a geração/seleção de casos de teste (MATHUR, 2008; DELAMARO et al., 2007; HIERONS et al., 2009; NETO, 2009; SAHIN; AKAY, 2016; ANAND et al., 2013; MOHI-ALDEEN et al., 2016). Enquanto que, na maioria das vezes, as técnicas de teste de software buscam particionar o domínio de entrada em subdomínios, saber qual elemento selecionar, para obter casos de teste com alta probabilidade de revelação de defeitos, é uma atividade complexa, dado a grande quantidade de entradas que o software pode processar, e mesmo devido às diferentes características dos produtos de software em domínios diversos.

O problema da geração de casos de teste para software é indecidível (DELAMARO et al., 2007). Ou seja, é um problema em que é impossível construir um algoritmo único capaz de responder com eficácia sempre a mesma questão. Nessa direção, uma série de esforços por parte de empresas/instituições ao redor do mundo são despendidos, a fim de se obter novas soluções para a geração/seleção de casos de Teste de Software, de forma a se ter *suites* de teste (conjuntos de casos de teste) que sejam menores (menor custo), mas que tenham alta capacidade de detectar defeitos no software (alta efetividade).

Em função dessa necessidade, muitas são as contribuíções para se obter conjuntos de casos de teste que sejam menores ou mínimos e, com isso, necessitar de menos tempo para executá-los (YOO; HARMAN, 2012; AHMED, 2016; HUANG et al., 2016; KHAN et al., 2016). Em resumo, esses esforços estão relacionados com o problema da minimização das *suites* de teste onde o objetivo é diminuir o tamanho desses conjuntos pela eliminação de casos de teste redundantes contidos no mesmo (YOO; HARMAN, 2012). Um dos métodos para se alcançar esse objetivo é via *Designs* Combinatoriais. *Designs* Combinatoriais faz parte da análise combinatória, cujo objetivo é preocupar-se com questões que tratam sobre a possibilidade de organizar elemen-

tos de um conjunto finito em subconjuntos, de modo que certas propriedades de balanço ou simetria sejam satisfeitas (STINSON, 2004).

Existem pesquisas que abordam a utilização de *Designs* Combinatoriais no contexto de Teste de Software, o que ficou conhecido como Teste Combinatorial. Tais abordagens têm chamado a atenção da comunidade de Teste de Software pois geram conjuntos de casos de teste menores (menor custo de execução) e eficiente (maior capacidade de encontrar falhas no software), onde foram eficazes na detecção de falhas devido à interação de várias variáveis de entrada (fatores). Por exemplo, em (KUHN et al., 2004) os autores afirmam que, tipicamente, quantidades consideráveis de falhas no software podem ser reveladas pela cobertura de todas as interações entre 4 e 6 fatores. Os conjuntos de casos de teste gerados via *Designs* Combinatoriais revelaram-se mais eficientes na detecção de falhas do que os conjuntos gerados por meio de técnicas de Teste de Software consagradas, tais como as técnicas da Caixa Preta e da Caixa Branca (TATSUMI, 1987).

1.1 Motivação

Em estudo publicado em (TATSUMI, 1987), conjuntos de casos de teste gerados por *Designs* Combinatoriais mostraram-se mais eficientes na revelação de defeitos do que técnicas de Teste de Software consagradas, como algumas técnicas de teste caixa preta e teste caixa branca. Nesse mesmo estudo, o uso de *Designs* Combinatoriais também mostrou-se mais fácil de se aplicar do que tais técnicas, uma vez que não exige extenso conhecimento do funcionamento do sistema por parte do projetista de teste de software, uma vez que para a sua aplicação, na maior parte das vezes, basta ter conhecimento dos parâmetros de entrada que envolvem o Sistema Sob Teste (SST).

Em (DALAL et al., 1999) e (TAI; LEI, 2002), foi demonstrado que conjuntos de casos de teste que cobrem todas as interações de parâmetros, dois a dois, podem revelar de 50% a 75% dos defeitos de um programa. Mais recentemente, (PETKE et al., 2015) mostram que o uso de *Designs* Combinatoriais é uma eficiente técnica para revelação de defeitos, via comparação entre diferentes *suites* de teste geradas por meio de *Simulated Annealing* (NURMELA; ÖSTERGåRD, 1993) e Algoritmos Gulosos (COHEN et al., 2014; LEI et al., 2007).

Diversos algoritmos existem para a geração de *Designs* Combinatoriais. Dentre eles, o *In-Parameter-Order-General* (IPOG-F) (FORBES et al., 2008) é um dos mais utilizados pela comunidade de testes de software . No entanto, o IPOG-F faz uso de

duas matrizes auxiliares para chegar à solução final, o que faz com que o algoritmo demande mais memória para executar. Além disso, tal abordagem realiza comparações exaustivas para estender cada caso de teste. Outro algoritmo muito popular é o *Tabu Search Approach* (TSA) (HERNANDEZ et al., 2010), que embora gere conjuntos de casos de teste menores que o algoritmo IPOG-F, requer a realização de diversos cálculos para chegar à solução final. Outro algoritmo bastante popular é o que está implementado na ferramenta *jenny* (JENKINS, 2016). No entanto, a *jenny* limita o tamanho da entrada que pode ser submetida ao algoritmo.

Desse modo, é importante pensar em novas soluções para gerar casos de teste via *Designs* Combinatoriais em que se busque olhar de uma maneira diferente para o problema, a fim de superar algumas limitações existentes em outras abordagens. Nesse sentido, um algoritmo que consiga gerar conjuntos de casos de teste menores para serem executados e que, mesmo assim, possua efetividade comparável a soluções já existentes na literatura é algo bastante motivador.

1.2 Objetivo e Metodologia de Pesquisa

O objetivo dessa dissertação de mestrado é apresentar uma nova forma de gerar conjuntos de casos de teste via *Designs* Combinatoriais, sendo que tais casos de teste tenham custo menor e efetividade comparável à soluções já existentes na literatura.

Desse modo, um algoritmo, denominado *T-Tuple Reallocation* (TTR; Realocação de T-Tuplas) (BALERA; SANTIAGO JÚNIOR, 2015; BALERA; SANTIAGO JÚNIOR, 2016), para gerar casos de teste de software via *Designs* Combinatoriais, especificamente via a técnica de Matriz de Cobertura com Níveis Variados (MCNV), foi desenvolvido. A ideia geral do TTR é derivar uma MCNV M por meio da criação e realocação de *t-tuplas* para a matriz M, considerando um parâmetro chamado *meta* (ζ). Tal parâmetro é calculado de acordo com o valor da combinação simples entre a quantidade de fatores selecionados, o *strength*, e a quantidade de t-tuplas que ainda não foram cobertas. As *t-tuplas* são realocadas permamentemente, à medida que fazem com que o caso de teste que as "hospeda" atinja a meta correspondente.

Três versões do algoritmo TTR foram desenvolvidas como resultado desse trabalho de pesquisa, e implementadas na linguagem de programação Java. A versão 1.0 é a versão original do TTR (BALERA; SANTIAGO JÚNIOR, 2015). Na versão 1.1 (BALERA; SANTIAGO JÚNIOR, 2016), foi feita uma mudança relacionada a não ordenação das variáveis de entrada. Já na última versão, 1.2, o algoritmo não gera mais a matriz Θ (matriz de t-tuplas), e sim, gera *t-tupla* por *t-tupla*, e a fórmula para calcular as

metas passa a considerar uma nova variável *booleana*.

Considerando o fato de que existem outros algoritmos/ferramentas tradicionais para Teste Combinatorial, cujo intuíto é a geração de MCNVs, é importante que se façam avaliacões de custo e efetividade rigorosas, comparando o algoritmo proposto nesse trabalho com as demais soluções. Portanto, é preciso demonstrar que o custo e/ou efetividade do mesmo é superior a alguns algoritmos/ferramentas significativos e existentes na literatura. Essa demonstração pode ser feita por meio de avaliações rigorosas, tais como experimentos controlados ou quasiexperimentos (LEMOS et al., 2013; ZANNIER et al., 2006), onde hipóteses são formuladas para uma avaliação estatística dos resultados, que fornecem uma maior credibilidade aos mesmos. Portanto, dois experimentos controlados e um quasiexperimento foram realizados para comparar o TTR com outros quatro algoritmos/ferramentas bastante conhecidos que geram MCNVs: IPOG-F (FORBES et al., 2008), *jenny* (JENKINS, 2016), IPO-TConfig (WILLIAMS, 2000), e *Pairwise Indepedent Combinatorial Testing* (PICT) (CZERWONKA, 2006). No primeiro experimento controlado, o intuíto era fazer uma comparação sob duas perspectivas de custo considerando a versão 1.1 do algoritmo TTR: tamanho das *suites* de teste e tempo para gerar as *suites* de teste. Além disso, foi realizada uma análise de similaridade entre as *suites* de teste, cujo objetivo é perceber se as mesmas sugeriam os mesmos casos de teste. No segundo experimento controlado, foi considerada a versão final do TTR, versão 1.2, comparando-a aos mesmos quatro algoritmos/ferramentas anteriores, mas considerando somente a perspectiva de custo relacionada ao tamanho das *suites* de teste e análise de similaridade.

Relativo à efetividade, um quasiexperimento foi realizado para comparar a versão 1.2 do algoritmo TTR com as outras mesmas quatro soluções, usando teste de mutação (DELAMARO et al., 2007; MATHUR, 2008) e considerando estudo de caso da área espacial (SANTIAGO JÚNIOR, 2011; SANTIAGO JÚNIOR; VIJAYKUMAR, 2012). Antecipadamente, pode-se afirmar que as conclusões dessas três avaliações rigorosas são que o TTR foi o algoritmo que apresentou melhor custo (menor quantidade de casos de teste para serem executados), mas que não há diferença de efetividade entre o TTR e as demais soluções. Além disso, as *suites* de teste não são similares, comparando o TTR com as outras soluções. Desse modo, pode-se afirmar que o TTR foi superior aos demais algoritmos/ferramentas pois teve mesma efetividade, porém melhor custo.

1.3 Organização do texto

A estruturação para esse trabalho de pesquisa é descrita a seguir:

(a) Capítulo 2. Esse capítulo apresenta a fundamentação teórica relacionada a essa dissertação de mestrado. Conceitos de teste de software, principais algoritmos/soluções para gerar *Designs* Combinatoriais, análise de mutantes, avaliações rigorosas (experimentos controlados, quasiexperimentos) são explorados;

(b) Capítulo 3. Esse capítulo apresenta, em detalhes, as três versões do algoritmo proposto, o TTR: 1.0, 1.1, e 1.2;

(c) Capitulo 4. Esse capítulo mostra avaliações rigorosas experimentais do TTR comparando-o com quatro outros algoritmos/ferramentas para teste combinatorial conhecidos na literatura: IPOG-F (FORBES et al., 2008), *jenny* (JENKINS, 2016), IPO-TConfig (WILLIAMS, 2000), e PICT (CZERWONKA, 2006);

(d) Capítulo 5. As consideracões finais são apresentadas nesse capítulo. Direções de pesquisa futuras para esse trabalho também são mencionadas;

(e) Apêndice A. Esse apêndice apresenta uma visão geral da arquitetura da implementação do algoritmo TTR em Java, por meio de Diagrama de Classes;

(f) Apêndice B. Esse apêndice mostra um breve manual do usuário para executar a implementação em Java do algoritmo TTR.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo apresenta a fundamentação teórica relacionada a esse trabalho de pesquisa. Teste de software na sua visão mais ampla (DELAMARO et al., 2007), geração de casos de teste via *designs* combinatoriais (MATHUR, 2008), Análise de Mutantes (DELAMARO et al., 2007; DEMILLO et al., 1978; JIA; HARMAN, 2011; CAMPANHA et al., 2010; WONG et al., 1995; DELAMARO, 1993) e avaliações rigorosas (experimentos controlados, quasiexperimentos) (CAMPANHA et al., 2010; ROTHERMEL et al., 2000; WOHLIN et al., 2000; LEMOS et al., 2013) são os temas abordados.

2.1 Teste de Software

O conceito de qualidade está relacionado ao atendimento dos requisitos pré-definidos no início do projeto. Dessa maneira, é importante identificar inconsistências inseridas no software que impeçam que tal objetivo seja atingido. Essas incosistências se arrastam por todo o ciclo de vida de desenvolvimento do software. Tais inconsitências são definidas a seguir (DELAMARO et al., 2007):

a) Defeito ou Falta: que consiste de um passo, processo, ou definição de dados feita de maneira incorreta, o que ocorre durante a implementação do software;

b) Erro: estado inesperado ou inconsistente, ocasionado por um defeito;

c) Falha: resultado produzido pela execução diferente do esperado. É a última etapa, ocorre na interface com o usuário.

Segundo (DELAMARO et al., 2007), o processo de Teste de Software é composto por um conjunto de atividades cuja finalidade é garantir que tanto o modo pelo qual o software está sendo construído, quanto o produto em si estejam de acordo com o especificado. Realizá-lo, porém, é uma tarefa complexa. Deve-se considerar uma série de variáveis que fogem do controle do testador como, por exemplo, a natureza distinta dos defeitos introduzidos no código. Por esse motivo, se torna necessário que o processo de Teste de Software seja dividido em fases com objetivos distintos (DELAMARO et al., 2007). A seguir, as fases gerais do processo de Teste de Software são mencionadas:

a) **Teste de Unidade:** foco nas menores unidades de um programa como funções, procedimentos, métodos ou classes. Nesse contexto, espera-se que

sejam encontrados defeitos relacionados a algoritmos mal implementados, estruturas de dados definidas de forma incorreta, ou simplesmente defeitos de programação;

b) **Teste de Integração:** deve ser executado após os testes de unidade terem sido finalizados completamente, a enfase é dada na construção da estrutura do sistema. À medida que as diversas partes do software são colocadas para trabalhar juntas, é preciso verificar interações existentes, verificando se as partes funcionam de maneira adequada e não levam a novos defeitos;

c) **Teste de Sistema:** é executado quando o sistema está todo desenvolvido. O objetivo é verificar se as funcionalidades especificadas estão todas corretamente implementadas. Resquisitos não funcionais são explorados nessa fase também;

d) **Teste de Aceitação:** o teste de aceitação é direcionado ao usuário final, que irá utilizar o sistema. Seu intuíto é servir para avaliar o nível de aceitação do sistema perante o cliente;

e) **Teste de Regressão:** o objetivo é verificar se a manutenção não inseriu novos defeitos.

Algumas definições importantes para o entendimento desse trabalho são dadas a seguir.

Definição 2.1. Um *dado de teste* é um elemento qualquer do domínio de entrada do software a ser testado, denominado *Software Under Test* (SUT).

Definição 2.2 Um *caso de teste* é um par formado por um conjunto de dados de teste e os resultados esperados[4]. Os casos de teste deverão ser executados, estimulando o SUT.

Definição 2.3 Um conjunto de casos de teste é uma conjunto de teste.

Testar um software, entre outras atividades envolvidas, requer submeter o produto a um conjunto de casos de teste específico e verificar se as saídas produzidas pelo software são as esperadas. Contudo, deve-se considerar que a execução de teste de

[4]Em *designs* combinatoriais, no contexto de geração de "casos" de teste, usualmente o que se gera são somente os dados de teste, omitindo-se os resultados esperados. Portanto, nesse trabalho está sendo, e continuará sendo usado, o termo "casos" de teste nesse sentido adotado em *designs* combinatoriais.

maneira exaustiva não é factível. Para tornar possível então o processo de teste, faz-se uso de uma série de técnicas e critérios de teste, cuja finalidade é a geração de um cojunto limitado de casos de teste.

Segundo (ROCHA et al., 2001), as técnicas e critérios de software fornecem ao testador uma abordagem sistemática e teoricamente fundamentada para a condução do processo de teste, sendo que, tais técnicas são diferenciadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste.

Tabela 2.1 - Adaptação da definição de (MATHUR, 2008) para a classificação das diferentes maneiras de geração de casos de testes (SANTIAGO JÚNIOR, 2011).

Categoria	**Origem**	**Técnica**	**Exemplo**
1	Requisitos (Informal)	Caixa-Preta	- Ad hoc - Análise do valor limite - Particionamento em classes de equivalência - Teste aleatório - *Designs* combinatoriais/ *t-way testing*
2	Requisitos (Formal)	*Baseado em modelos*	- Baseado em *Statecharts* - Baseado em Máquinas de Estados Finitos - Baseado em B - Baseado em Z
3	UML	Baseado em documentos UML	-Baseado em UML (Máquinas de Estados, Diagramas de Classes, Sequencia, etc.)
4	Código-fonte	Caixa-Branca	- Teste de Fluxo de Controle - Teste de Fluxo de Dados - Teste de Mutação

Definição 2.4. Um *cenário de teste* é definido como uma interação entre o usuário e o *Software Under Test* (SUT).

A Tabela 2.1 relaciona uma boa parte das principais técnicas para geração de casos de teste de software existentes na literatura, com alguns de seus exemplos mais populares. A cada técnica de teste também é relacionada a sua origem, ou seja, o artefato que é a base para que os casos de teste sejam gerados. Percebe-se que o conjunto de casos de teste pode ser gerado a partir de especificações formais ou informais, com ou sem conhecimento do código fonte (SANTIAGO JÚNIOR, 2011). *Designs* combinatoriais/*t-way testing* (definição na Seção 2.2) podem ser tidos como um exemplo de teste caixa preta, pois não precisa-se ter conhecimento do código fonte para gerar casos de teste. Além disso, tais técnicas podem ser usadas não para gerar os casos mas, sim, para gerar os cenários de teste (SANTIAGO JÚNIOR, 2011).

Dessa maneira, as técnicas de teste de software buscam estabelecer o contexto no qual a atividade de teste vai ocorrer, ou seja, os testes vão ser feitos com base em que tipo de artefato (formal, informal, etc.). Já os critérios de teste, segundo (DELAMARO et al., 2007), são um cojunto de regras para particionar o domínio de entrada em subdomínios, para a partir deles, os dados de teste serem extraídos.

Contudo, a maioria das técnicas e critérios de teste acaba por se limitar a particionar o domínio de entrada em sudomínios, mantendo a questão da escolha dos pontos que possuem a maior probabilidade de revelar defeitos, ainda sem resposta única. Tal escolha compete à atividade de geração de **casos de teste**.

A próxima seção fornecerá uma visão geral sobre *designs* combinatoriais.

2.2 *Designs* Combinatoriais

Segundo (MATHUR, 2008), *designs* combinatoriais são um conjunto de técnicas de geração de casos de teste de software que buscam a seleção de um pequeno número de casos de teste, uma vez que o domínio de entrada e o número de subdomínios em suas partições é largo e complexo.

Dado um programa P, chama-se cada variável de entrada de *fator* e os valores associados a cada fator são conhecidos como *níveis*. O conjunto de níveis um para cada fator, é chamado de *combinação de fatores*, $n \in N$. Portanto, N é o conjunto conjunto de combinações de fatores (SANTIAGO JÚNIOR, 2011). *Strength*, t, é o grau de interações entre os fatores. Por exemplo, no *pairwise testing*, o grau de interação é dois, portanto, o valor do *strength* é 2. No *t-way testing*, uma *t-tupla* é uma interação de fatores e níveis de tamanho igual ao *strength*. Portanto, uma *t-tupla* é uma lista ordenada de elementos.

As técnicas de *designs* combinatoriais podem ser classificadas em *designs* balanceados e *designs* não-balanceados. Matrizes Ortogonais (MO) e Matrizes Ortogonais com Níveis Variados (MONV) são exemplos de *designs* balanceados. Tais técnicas exigem que todo vetor $|N| \times t$ contenha cada *t-tupla* exatamente o mesmo número de vezes. Em teste de software, esse requisito de balanço nem sempre é essencial a menos que o SUT seja conhecido por comportamento não-determinístico (MATHUR, 2008). Já as técnicas Matriz de Cobertura (MC) e Matriz de Cobertura com Níveis Variados (MCNV) são exemplos de *designs* não-balanceados, em que que todo vetor $|N| \times t$ as *t-tuplas* não aparecem exatamente o mesmo número de vezes. i.e., cada *t-tupla* aparece no mínimo uma vez. Em outras palavras, significa que a ocorrência do número de *t-tuplas* pode variar. Essa diferença frequentemente conduz a *Designs* Combinatoriais de menor tamanho se comparadas ambas as classes ($|CA| < |OA|$; $|MCA| < |MOA|$)

MCNV é uma das mais populares técnicas da comunidade de teste de software. Em uma MCNV, é possível que fatores (parâmetros) tomem níveis (valores) de diferentes conjuntos. Agora, a definição formal de uma MCNV.

Definição 2.5 Uma MCNV é uma matriz definida por MCNV(N,$l_1^{k_1}$ $l_2^{k_2}$...$l_p^{k_p}$,t), onde N é a quantidade de linhas da matriz onde o fator k_1 tem l_1 níveis, ..., e o fator k_p tem l_p níveis. O parâmetro t corresponde ao grau de interação das variáveis, ou *strength*. No contexto de teste de software, cada combinação de fatores (ou seja, cada linha da MCNV) é um caso de teste. Dessa forma, a MCNV é uma conjunto de teste.

Dessa forma, uma MCNV é gerada por *t-way testing*. Por exemplo, o clássico *2-way testing* (ou, *pairwise testing*), produz uma MCNV para $t = 2$. Para se ter uma idéia da redução do conjunto de casos de teste obtida via *designs* combinatoriais, considere que existam 10 fatores $(A, B, \cdots, J)$ e que cada fator possua 5 níveis, ou seja, $A = \{a_1, a_2, \cdots, a_5\}$, $B = \{b_1, b_2, \cdots, b_5\}$, ..., $J = \{j_1, j_2, \cdots, j_5\}$. Se uma combinação exaustiva fosse realizada, teria-se um total de $5^{10} = 9.765.625$ casos de teste da forma $ct = \{a_k, b_k, \cdots j_k\}$. Utilizando um algoritmo para gerar *designs* combinatoriais, como o algoritmo TTR (BALERA; SANTIAGO JÚNIOR, 2015; BALERA; SANTIAGO JÚNIOR, 2016) apresentado nesse trabalho, com grau de interação t igual a 2, teria-se 48 casos de teste.

2.2.1 Principais Algoritmos/Ferramentas para Geração de *Designs* Combinatoriais

Nessa seção, é apresentado alguns estudos relevantes na área de *Designs* Combinatoriais. Em (HERNANDEZ et al., 2010), é proposto um algoritmo para a geração de MCNVs utilizando a meta-heurística busca tabu (*Tabu Search Approach* (TSA)). O princípio fundamental de funcionamento do algoritmo funciona em torno da seleção entre 3 funções de vizinhança a cada momento, através de uma determinada probabilidade, ou seja, ao invés da utilização de apenas uma função, o algoritmo pode escolher dentre três. Essa probabilidade é determinada a partir de um processo de ajuste fino ao longo de um conjunto de probabilidades discretas. Tal característica demanda muita memória para o processamento do algoritmo, além do fato de que, para extender cada caso de teste, o algoritmo precisa realizar o cálculo de probabilidades para a escolha da função que irá utilizar.

In-Parameter-Order General (IPOG-F) (FORBES et al., 2008) é um algoritmo para a geração de MCNVs, que consiste de uma adaptação da estratégia *In-Parameter-Order* (IPO) (LEI et al., 2007) para *strength* superior a 2. Por meio de dois passos principais, o *crescimento horizontal* e o *crescimento vertical*, que trabalham em cima de uma solução inicial, a MCNV é construída. Para isso, o algoritmo conta com o apoio de duas matrizes auxiliares. O uso de matrizes auxiliares encarece o processo como um todo, uma vez que demanda mais memória para a sua execução (armazenamento, gasto com acessos desnecessários, etc.). Além disso, o algoritmo realiza comparações exaustivas a cada extensão horizontal, o que acaba levando a uma execução mais demorada do algoritmo.

Automatic Efficient Test Generator (AETG) (COHEN et al., 2014), é uma ferramenta para a geração de casos de teste de forma automática, que reúne algoritmos que suportam o *t-way testing*. O algoritmo implementado gera os casos de teste por meio de *Experimental designs* (ED), que são técnicas estatísticas utilizadas para planejar experiências de maneira que se possa extrair o máximo de informação possível por meio da menor quantidade de experimentos possível. É um algoritmo guloso, em que cada linha de teste é construída de cada vez, i.e., ele não trabalha utilizando uma solução inicial.

O algoritmo implementado na ferramenta *Pairwise Independent Combinatorial Testing* (PICT) (CZERWONKA, 2006) tem duas fases principais: preparação e geração. Na primeira fase, o algoritmo gera todas as *t-tuplas* a serem cobertas. Na segunda fase, é responsável pela geração do MCNV propriamente dito, através do uso da

heurística gulosa. É possível entender que o algoritmo PICT é parecido com o algoritmo AETG, com alguma espécie de mecanismo de otimização de velocidade. No entanto, no estudo não fica muito claro a maneira com que o algoritmo está implementado. Além disso, o algoritmo foi implementado para executar somente na plataforma *Windows*.

A ferramenta *Advanced Combinatorial Testing System* (ACTS) (YU et al., 2013a) estão implementados diversos algoritmos, IPOG(LEI et al., 2007), IPOG-D(LEI et al., 2007), IPOG-F(FORBES et al., 2008), IPOG-F2(FORBES et al., 2008) e PaintBall, sendo o IPOG o algoritmo principal da ferramenta. De acordo com a ordem de complexidade das instâncias de teste submetidas a ferramenta, e as características de cada algoritmo implementado, a ferramenta escolhe qual é o algoritmo mais adequado para aquela instância de teste. Tal ferramenta carrega com si as limitações dos algoritmos implementados nela, como por exemplo, as descritas acima, referente ao algoritmo IPOG-F. Além disso, existe gasto de recursos relacionado à escolha do algoritmo adequado para o problema do usuário.

Em (CAVALGNA et al., 2013), é apresentada uma nova ferramenta para a geração de MCNVs com suporte a tratamento de restrições, o CitLab. Assim como a ferramenta ACTS, ela também é composta por uma série de algoritmos existentes na literatura para a geração de MCAs, cada um com as suas peculiaridades. São implementados os algoritmos AETG, IPO e ACTS, dentre outros. No fundo, a geração de casos de teste com MCNVs é apenas uma das características da ferramenta proposta, dessa forma, a preocupação com o custo relacionado a esse aspecto não é levada em conta. Assim como o ACTS, a ferramenta proposta em (CAVALGNA et al., 2013) carrega não só as desvantagens de algoritmos já existentes na literatura, como também, o gasto com relação à escolha do algoritmo adequado para o uso.

Em (HENARD et al., 2014), propõe-se um novo algoritmo para a geração de *designs* combinatoriais, a partir da técnica baseada em busca. Uma heurística de similaridade é utilizada para a escolha das melhores configurações. Tal algoritmo considera a presença de restrições[1] durante a geração dos dados, e por isso, é considerado uma evolução dos algoritmos tradicionais. Seu funcionamento se resume a dois passos: (i) geração de configurações válidas usando técnicas de busca e (ii) a seleção de configurações de acordo com o critério de teste *t-way testing*, através da heurística de similaridade. No entanto, o algoritmo é independente do *strength*, dessa maneira,

[1]Restrições seriam configurações que não fazem sentido para um determinado sistema. Dessa maneira, *t-tuplas* que incluem tais configurações não devem ser consideradas na geração de MCNVs.

nem sempre ele cobre todas as interações de fatores.

O estudo feito em (YU et al., 2013b) tem por objetivo propor o algoritmo IPOG-C para a geração de MCNVs considerando a presença de restrições. O algoritmo foi implementado na ferramenta ACTS, e é uma adptação do algoritmo IPOG para que haja suporte a restrições, através de um solver SAT[2] . A maior contribuíção do algoritmo proposto são três otimizações que podem ser aplicadas a outros algoritmos para a geração de casos de teste. Tais otimizações buscam a redução do número de chamadas do solver SAT. O algoritmo é uma adaptação do IPOG, dessa maneira, assim como ele, realiza comparações exaustivas a cada extensão horizontal, o que acaba levando a uma execução mais demorada do algoritmo, além do fato de que cada *t-tupla* é avaliada, para verificar se é valida ou não.

Em (YILMAZ et al., 2014), o estudo tráz um novo algoritmo para a geração de *designs combinatoriais* chamado *Feedback Driven Adptative Combinatorial Testing Process* (FDA-CIT). A cada iteração do algoritmo, é verificado o potencial de mascaramento de falhas, isolando as suas causas prováveis e então gerando uma nova configuração que omite tais causas. Resultados sugerem que efeitos mascarados existem na prática, e que o algoritmo proposto prove uma eficiente maneira de trabalhar com esse problema, sem exigir que esse tipo de efeito seja conhecido antes da execução dos testes. No entanto, não há preocupações com o custo do algoritmo associado à geração de MCNVs.

Em (PETKE et al., 2015), é feita uma comparação entre *Simulate Anneling* (SA), algoritmos gulosos e Algoritmos Genéticos (AG) no contexto de Teste Combinatorial. De fato, os autores não apresentam um novo algoritmo mas sim uma avaliação rigorosa de eficiência, como proposto nesse trabalho de dissertação. Os autores encontraram evidências para mudar algumas verdades sobre avaliação das abordagens: as restrições do mundo real permitem que SA atinja *strengths* de valores altos (a verdade é que SA não é escalável em relação a t). Não foi encontrada nenhuma evidência que sugere que algoritmos gulosos são menos efetivos nesse quesito comparado com SA: algoritmos gulosos superaram levemente SA. Contudo, AG não é escalável para altos valores de *strength*.

De maneira a comparar de maneira mais clara os estudos apresentados com o algoritmo proposto nesse trabalho de dissertação, o TTR, na Tabela 2.2 é mostrado

[2]SAT é sigla de SATISFIABILITY, que é um cojunto de problemas da ciência da computação cujo objetivo é determinar se existe uma interpretação que satisfaz uma determinada fórmula booleana.

algumas características principais de todos esses algoritmos/ferramentas. Nessa tabela, * significa que a característica está presente, - significa que a característica não está presente, e vazio (espaço em branco) significa que não está totalmente claro que a abordagem em questão tem a característica ou se é aplicável a ela.

Tabela 2.2 - Algoritmos/Ferramentas para Teste Combinatorial. Legenda: 1 = novo algoritmo, 2 = não mais que uma matriz auxiliar, 3 = avaliado via experimentos controlados/quasiexperimentos, 4 = não geração da matriz de *t-tuplas* completa, 5 = otimização do tempo, 6 = trabalha sobre conjunto de casos de teste inicial, 7 = não impõem limites de execução, 8 = apoio a *t-way testing*.

algoritmos/ferramentas	1	2	3	4	5	6	7	8
TSA(HERNANDEZ et al., 2010)	-	-	-	-		-		*
IPOG-F(FORBES et al., 2008)	*	-	-	-		*		*
AETG(COHEN et al., 2014)	-		-	-				*
PICT(CZERWONKA, 2006)	*		-	-	*			*
ACTS(YU et al., 2013a)	-	-	-	-		-		*
CitLab(CAVALGNA et al., 2013)	-		-	-				*
Similarity(HENARD et al., 2014)	-			-		-		*
IPOG-C(YU et al., 2013a)	*	-		-		*		*
FDA-Cit(YILMAZ et al., 2014)	*		-	-				
jenny(JENKINS, 2016)	*		-			-	*	*
SA/greedy/GA(PETKE et al., 2015)	-		*	-				*
TTR - 1.2	*	*	*	*	-	*	*	*

2.3 Análise de Mutantes

A técnica de Teste Baseada em Defeitos tem como objetivo definir seus critérios com base em defeitos que são frequentemente inseridos pelos programadores durante a fase de desenvolvimento do software e possíveis abordagens que possam detectá-los. Um dos mais famosos critérios dessa técnica é a Análise de Mutantes (DEMILLO et al., 1978).

Segundo (DELAMARO et al., 2007), o critério Análise de Mutantes considera defeitos típicos do processo de implementação de software. No fundo, seu objetivo é incluir ao código original (programa P), sendo o mesmo já testado e aprovado por uma conjunto de teste T, os erros sintáticos mais frequentes, um de cada vez, gerando "versões" similares de P, os chamados mutantes. Tais erros sintáticos, que podem ser por exemplo um incremento de variável, são incluídos por meio de operadores de mutação, que tem por finalidade gerar um mutante com um erro específico através

do código original de P. Entende-se por operador de mutação as regras que definem as alterações que devem ser aplicadas no programa original P. Por exemplo, a Figura 2.2 mostra um mutante do código original, representado na Figura 2.1. Repare que há uma sutil diferença entre os dois: a variável *dado* está sendo incrementada no mutante, enquanto que no código original, é apenas acessado o seu valor atual.

Figura 2.1 - Código original.

```
public  boolean inserir( float dado )
{
    int posicaoInicial;
    try {
        System.out.println( "#### Inserting data ..." );
        posicaoInicial = buffer.position();
        System.out.println( "#### Data: " + dado );
        buffer = buffer.putFloat( dado );
        System.out.println( "#### Data in position 1 of the buffer: " + buffer.getFloat( 4 ) );
        m_ixwr = buffer.position();
    } catch ( java.lang.Exception e ) {
        System.out.println( "#### Error inserting data!" );
        return false;
    }
    return true;
}
```

Figura 2.2 - Mutante.

```
public  boolean inserir( float dado )
{
    int posicaoInicial;
    try {
        System.out.println( "#### Inserting data ..." );
        posicaoInicial = buffer.position();
        System.out.println( "#### Data: " + ++dado );
        buffer = buffer.putFloat( dado );
        System.out.println( "#### Data in position 1 of the buffer: " + buffer.getFloat( 4 ) );
        m_ixwr = buffer.position();
    } catch ( java.lang.Exception e ) {
        System.out.println( "#### Error inserting data!" );
        return false;
    }
    return true;
}
```

Em seguida, os mutantes são executados pela mesma conjunto de teste T, e então é contabilizado quais mutantes foram considerados diferentes do programa original P, sendo esses denominados mutantes mortos, e quais mutantes foram considerados iguais, ou seja, que passaram pela conjunto de teste T, sendo esses denominados

mutantes equivalentes. O ideal é que todos os mutantes gerados sejam mortos, o que signifca que a conjunto de teste T é adequada ao programa P (DELAMARO et al., 2007).

No entanto, na grande maioria das vezes, o testador precisa lidar com os mutantes que sobreviveram a conjunto de teste T, de modo a identificar a existencia de mutantes equivalentes. Tal tarefa é uma questão indecidível, uma vez que depende de análise humana para verificar se aquele determinado mutante que sobreviveu é realmente um mutante equivalente ao código original ou que a conjunto de teste falhou em detectar aquele defeito específico (DELAMARO et al., 2007).

Segundo (DEMILLO et al., 1978), a Análise de Mutantes fornece uma medida objetiva do nível de confiança da adequação dos casos de teste analisados por meio da definição de um escore de mutação, que relaciona o número de mutantes mortos com o número de mutantes gerados. O escore de mutação é calculado da seguinte forma:

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

sendo:

- DM(P, T): número de mutantes mortos pelos casos de teste em T;
- M(P): número total de mutantes gerados;
- EM(P): número de mutantes gerados equivalentes a P.

O escore de mutação varia entre 0 e 1 sendo que, quanto maior o escore mais adequado é o conjunto de casos de teste para o programa sendo testado (DELAMARO et al., 2007). No fundo, o escore de mutação é a métrica que indica a qualidade da conjunto de teste, através da relação entre o número de falhas detectadas e o número total de falhas introduzidas (JIA; HARMAN, 2011).

O critério Análise de Mutantes pode ser aplicado a nível de unidade, nível de integração e nível de sistema. Pode ser aplicado a várias linguagens de programação, através do teste de unidade da técnica caixa-preta, como por exemplo Fortran, C, C#, e até mesmo SQL (JIA; HARMAN, 2011).

O critério Análise de Mutantes destaca-se por apresentar alta eficiência em revelar defeitos quando comparado a outros critérios (JIA; HARMAN, 2011; CAMPANHA et

al., 2010; WONG et al., 1995); ele ainda emprega um trabalhoso processo de implementação, uma vez que a sua utilização requer a intervenção humana na fase de identificação de mutantes equivalentes, já que o número de mutantes sobreviventes geralmente é alto e sua avaliação é complexa, fazendo com que a sua análise demande um alto custo.

Nesse contexto, existem à disposição uma série de ferramentas para apoiar o critério Teste de Mutação, as quais geralmente são exclusivas a determinadas linguagens, ou mesmo a paradigmas de programação. É o caso da ferramenta Proteum (DELAMARO, 1993) desenvolvida para apoiar o teste de mutação em programas escritos em linguagem C, da ferramenta Monthra (CHOI et al., 1989) para apoiar programas escritos em Fortran, e a MuJava (OFFUTT et al., 2006), para apoiar programas escritos em linguagem Java.

2.4 Avaliações Rigorosas

Existem muitos tipos de estudos sobre avaliações rigorosas que podem ser aplicados à pesquisa de Engenharia de Software (LEMOS et al., 2013). Tais métodos geralmente diferem entre si de acordo com o grau de rigor empregado no experimento ou características como a quantidade de amostras envolvidas. (ZANNIER et al., 2006) classifica os métodos de avaliação empírica em: Experimento Controlado, Quasi-Experiência, Estudo de Caso, Estudo de Caso Exploratório, Relatório de Experiência, Meta-Análise, Exemplo de Aplicação, Levantamento e Discussão. Em função da natureza do trabalho de pesquisa apresentado nessa dissertação, será dado atenção apenas aos Experimentos Controlados e Quasiexperimentos.

Segundo (LEMOS et al., 2013; ZANNIER et al., 2006) os experimentos controlados têm como principais características: a aplicação de atribuíção aleatória no tratamento das amostras, envolvem uma quantidade relativamente grande de amostras - geralmente mais que 10 amostras, formulam hipóteses, selecionam uma variável independente e às vezes aplicam amostragem aleatória, enquanto quasiexperimentos são experimentos controlados com uma ou mais das suas características ausentes.

No contexto da engenharia de software, os experimentos geralmente são usados para comparar diferentes abordagens, com relação à sua eficácia ou esforço de aplicação (LEMOS et al., 2013). Para isso, os experimentos controlados geralmente fazem uso do teste de hipótese, para verificar se as diferenças observadas são de fato significativas.

O Teste de Hipóteses é um método que faz parte da Inferência Estatística, cujo

objetivo é fornecer informações suficientes para refutar ou aceitar hipóteses formuladas em torno de um determinado experimento. A natureza do experimento vai determinar a maneira com que o teste de hipótese deve ser conduzido.

De acordo com a revisão bibliográfica realizada, não foi possível encontrar nenhum estudo que apresente um experimento controlado, considerando *designs* combinatoriais. A seguir, apresentamos alguns estudos relevantes que consideram experimentos controlados.

Em (LEMOS et al., 2013), os autores apresentaram uma perspectiva histórica de 25 anos de estudos de avaliação relacionados ao teste de software e que foram publicados no Brasil, representados pelo SBES e no mundo, representados pelo ICSE. Os principais resultados mostraram que a comunidade SBES tem aumentado consideravelmente os esforços na realização de avaliações rigorosas. No entanto, os autores afirmaram que esses esforços ainda são baixos em comparação com o número de estudos publicados no ICSE e que apresentam avaliações rigorosas. Além disso, concluíram que apenas um único artigo, em cada conferência, apresentava um experimento controlado. Estes dois estudos são descritos abaixo.

Em (CAMPANHA et al., 2010), foi apresentado um experimento controlado comparando custo e dificuldade de satisfação dos critérios de análise de mutação considerando os paradigmas de programação Procedural e Orientado a Objetos. Os autores avaliaram 32 programas desenvolvidos em C e Java. Eles usaram duas ferramentas, Proteum (DELAMARO, 1993) para código C, e MuClipse (OFFUTT et al., 2006) para código Java. O experimento foi conduzido com base na abordagem proposta em (WOHLIN et al., 2000). Os resultados mostraram que não só o custo, mas também a dificuldade de satisfação do paradigma Procedural são maiores do que o paradigma Orientado a Objetos.

Um experimento controlado comparando a metodologia de teste "*What You See is What You Test)*" (WYSIWYT) com uma abordagem *Ad Hoc* foi apresentado em (ROTHERMEL et al., 2000). Os resultados mostraram que os indivíduos que usaram a metodologia WYSIWYT realizaram testes significativamente mais eficazes, como medido pelo método de du-adequação, e foram avaliados como testadores mais eficientes, medido pela velocidade e redundância, do que aqueles que seguiram a abordagem *Ad Hoc*. Além disso, os indivíduos que seguiram WYSIWYT são menos confiantes que o *Ad Hoc*. Esses resultados são interessantes porque podem indicar que é possível obter alguns benefícios das noções formais de teste sem, de fato, treinamento formal relacionado ao teste.

2.5 Considerações Finais Sobre esse Capítulo

Esse capítulo apresentou a fundamentação teórica relacionada a essa dissertação de mestrado. As principais áreas de conhecimento relacionadas a esse trabalho são teste de software e *designs* combinatoriais. Portanto, foi apresentada uma visão geral sobre o processo de teste de software, com maior ênfase a adaptação de análise combinatória no contexto de teste de software, ou seja, geração de *designs* combinatoriais para teste (MCNVs via *t-way testing*). Teste de mutacão também foi mencionado pois foi usado para apoiar a comparação de eficiência do algoritmo TTR com outras soluções existentes na literatura (vide Seção 4.3). Como foram realizadas comparações rigorosas, via experimentos controlados e quasiexperimento, entre o TTR e outros quatro algoritmos/ferramentas que geram *designs* combinatoriais, então uma breve introdução sobre o assunto também foi salientada.

O próximo capítulo apresenta o algoritmo TTR em suas três versões.

3 TTR: UM ALGORITMO PARA TESTE COMBINATORIAL

Esse capítulo apresentará as três versões do algoritmo TTR (*T-Tuple Reallocation*), a principal contribuíção dessa pesquisa. Primeiramente, será apresentada a versão inicial, 1.0, do algoritmo TTR (BALERA; SANTIAGO JÚNIOR, 2015; BALERA; SANTIAGO JÚNIOR, 2016). Na versão 1.1, foi feita uma mudança relacionada a não-ordenação dos fatores de entrada: o algoritmo gera o conjunto de teste independe da ordem de entrada dos fatores, o que soluciona o problema da geração de conjunto de teste de tamanhos diferentes dependendo da ordem em que se apresenta os fatores ao algoritmo. Já a última versão, 1.2, o algoritmo constrói o conjunto de teste sem precisar partir da pior solução, ou seja, não gera mais a matriz completa de *t-tuplas*. As três versões do algoritmo TTR foram implementadas em linguagem de programação Java. O Apêndice A mostra uma visão em alto nível da arquitetura do software que implementa o algoritmo TTR (versão 1.2). O Apêndice B é um breve manual do usuário dessa ferramenta.

Antes de entrar em detalhes sobre cada uma das versões, é importante que alguns conceitos comuns a todas elas sejam definidos préviamente. Para isso, será feito uso do exemplo proposto na Figura 3.1, considerando o grau de interação, $t = 2$. É importante notar que neste exemplo, o teste é feito a nível de unidade, por isso, cada um dos parâmetros referentes *metodoQualquer* irá dar origem a um fator. Portanto, para esse exemplo, deve-se considerar 3 fatores: *x1*, *x2* e *x3*. Cada um desses fatores pode assumir uma determinada quantidade de valores, por exemplo, um fator do tipo *booleano* pode assumir no máximo dois valores: *verdadeiro* ou *falso*. A cada um desses valores correspondente a cada fator pode assumir dá-se o nome de *nível*. Dessa forma, considerando o exemplo da Figura 3.1, serão considerados o fator *x1*, que compreende os níveis *verdadeiro* e *falso*, o fator *x2*, que também compreende os níveis *verdadeiro* e *falso*, e por fim, o fator *x3*, que compreende os níveis *tipo1*, *tipo2* e *tipo3*. Por meio da técnica *t-way testing*, todas as t-tuplas deverão ser contempladas no conjunto de teste final, uma vez que tal técnica se baseia na idéia de que a maioria das falhas pode ser revelada a partir da interação de t fatores. Então, para este exemplo, a Tabela 3.1 ilustra a relação entre fatores e seus respectivos níveis.

Figura 3.1 - Assinatura de um método qualquer

```
public void metodoQualquer(boolean x1, boolean x2, ClasseQualquer x3){}
```

Tabela 3.1 - Exemplo de fatores e níveis

Fator	**Nível**
x1	*verdadeiro*, $falso$
x2	*verdadeiro*, $falso$
x3	*tipo1*, *tipo2*, *tipo3*

T-way testing é a cobertura de todas as interações entre os fatores e níveis associados ao *strength* t. A cada uma dessas combinações dá-se o nome de $t-tupla$. Tomando ainda o exemplo da Figura 3.1 e considerando, $t = 2$, uma $t-tupla$ seria composta, por exemplo, do primeiro nível do primeiro fator juntamente com o primeiro nível do segundo fator. No algoritmo proposto neste trabalho, as $t-tuplas$ serão armazenadas em uma matriz chamada Θ, representada na Tabela 3.2. Repare que na tabela, todas as *t-tuplas* foram feitas a partir da combinação 2 a 2 de cada um dos fatores, por conta de t.

Tabela 3.2 - Matriz Θ para o exemplo da Figura 3.1

i	**x1**	**x2**	**x3**
1	*verdadeiro*	*verdadeiro*	
2	*verdadeiro*	$falso$	
3	$falso$	*verdadeiro*	
4	$falso$	$falso$	
5	*verdadeiro*		*tipo1*
6	*verdadeiro*		*tipo2*
7	*verdadeiro*		*tipo3*
8	$falso$		*tipo1*
9	$falso$		*tipo2*
10	$falso$		*tipo3*
11		*verdadeiro*	*tipo1*
12		*verdadeiro*	*tipo2*
13		*verdadeiro*	*tipo3*
14		$falso$	*tipo1*
15		$falso$	*tipo2*
16		$falso$	*tipo3*

Definição 3.5: Matriz Θ é uma matriz que armazena uma coleção de *t-tuplas*, definda por $\Theta = \{\theta_i | i = 1...m\}$, onde θ_i é uma das *t-tuplas* formadas e m é a quantidade total delas. Observe a Tabela 3.2 em que as *t-tuplas* são formadas a partir das interações de fatores, de tamanho t. É importante ressaltar que, a cada uma das *t-tuplas* será associado um parâmetro chamado *flag*, que será melhor explicado a

diante.

A partir daí, todas as $t-tuplas$ deverão estar na matriz de testes final, a matriz M. A Figura 3.3 mostra um exemplo de matriz de testes que satisfaz essa condição: repare que o primeiro caso de teste (τ_1) contempla as tuplas θ_1, θ_4 e θ_{10} da Tabela 3.2.

Tabela 3.3 - Matriz M de testes para o exemplo da Figura 3.1

x1	**x2**	**x3**
verdadeiro	*verdadeiro*	*tipo1*
verdadeiro	$falso$	*tipo2*
verdadeiro	*verdadeiro*	*tipo3*
$falso$	$falso$	*tipo1*
$falso$	*verdadeiro*	*tipo2*
$falso$	$falso$	*tipo3*

Definição 3.6: $M_{|n|\times|(f+1)|}$ é um MCNV que guarda todos os casos de teste, definido por $M = \{\tau_i | i = 1...|n|\}$, onde τ_i representa um caso de teste e $|n|$ é a quantidade de casos de teste gerados pelo algoritmo e f a quantidade de fatores. A $(f + 1)$-ésima coluna não é usada para representar nenhum fator, mas sim o valor de ζ associado a cada caso de teste. Considerando o exemplo da Figura 3.1, $M = \{\tau_1 = \{verdadeiro, verdadeiro, tipo1\}$, $\tau_2 = \{verdadeiro, falso, tipo2\}$, $\tau_3 = \{verdadeiro, verdadeiro, tipo3\}$, $\tau_4 = \{falso, falso, tipo1\}$, $\tau_5 = \{falso, verdadeiro, tipo2\}$, $\tau_6 = \{falso, falso, tipo2\}$ $\}$. A cada um dos casos de teste, será associado o parâmetro ζ, que corresponde a *meta*, que será melhor explicado a seguir.

3.1 TTR: Versão 1.0

O TTR gera uma MCNV por meio de *t-way testing*. Uma visão geral da versão 1.0 do algoritmo TTR é vista no Algoritmo 1. O conceito geral do TTR é a construção de uma matriz, M, por meio da realocação das *t-tuplas* da matriz Θ para a matriz M, e então cada *t-tupla* realocada deve cobrir a maior quantidade de *t-tuplas* ainda não cobertas em Θ até o momento e considerando um parâmetro denominada *meta* (ζ). Esse parâmetro está relacionado a um objetivo para cada linha da matriz M. Cada *meta* é calculada por meio da combinação simples de t e a quantidade de fatores cobertos pela *t-tupla* em um determinado momento.

Algorithm 1 Visão geral: TTR 1.0

entrada: $f = \{f_i \mid i = 1...|f|\}, t$
saída: $M_{|n|\times|(f+1)|} = \{\tau_i \mid i = 1..|n|\}$

1: $f \leftarrow ordenar(f)$
2: $\Theta \leftarrow Construtor(f, t)$
3: **while** $\Theta \neq \emptyset$ **do**
4: $\quad M \leftarrow calcularSolucaoInicial(\Theta)$
5: $\quad \zeta \leftarrow calcularZeta(M)$
6: $\quad [M, \Theta] \leftarrow Principal(M, \Theta, \zeta)$

T-way testing é a cobertura de todas as interações de níveis e fatores associados ao *strength t*. Então, o algoritmo TTR segue os seguintes passos: (i) gerar todas as possíveis *t-tuplas* que ainda não foram cobertas. O procedimento *Construtor* constrói a matriz Θ; (ii) gerar uma solução inicial, a matriz M; e (iii) realocar as *t-tuplas* com o objetivo de chegar à melhor solução final por meio do procedimento *Principal*. Então, o conjunto final de casos de testes é atualizado na matriz M. Esses passos são detalhados a seguir, fazendo uso do exemplo apresentado na Figura 3.2, onde os níveis e fatores estão apresentados na Tabela 3.4 e $t = 2$.

Figura 3.2 - Assinatura do método cadastrar

```
public void cadastrar(Banco banco, Funcão função, Bandeira bandeira){}
```

Tabela 3.4 - Exemplo de fatores e níveis

Fator	**Nível**
banco	*BancoA*, *BancoB*
função	*debito*, *credito*
bandeira	*CartaoA*, *CartaoB*, *CartaoC*

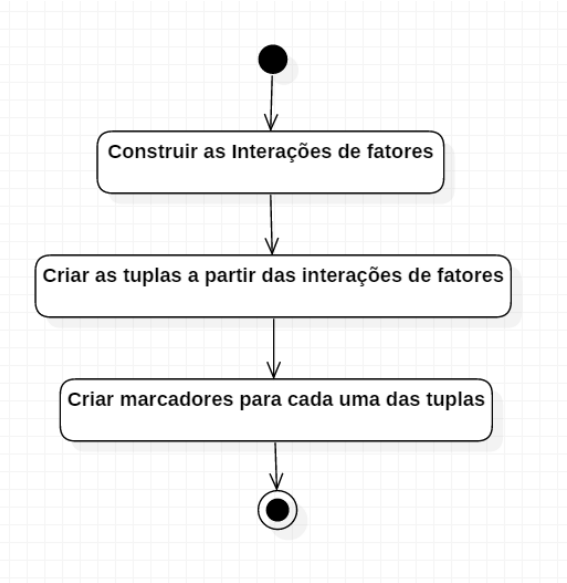
3.1.1 O Procedimento *Construtor*

De acordo com a entrada especificada (fatores e níveis), o procedimento *Construtor* tem como objetivo gerar todas as *t-tuplas* que serão cobertas. Cada *t-tupla* é

representada na matriz $\Theta_{|C|\times|f|}$ [5], onde $|C|$ representa o número de *t-tuplas*, t é o *strength* e $|f|$ é o número de fatores.

Cada linha, θ_i, de Θ é uma *t-tupla* que ainda não foi coberta e possui um parâmetro, *flag*, associado a ele cujo objetivo é auxiliar no processo de realocação das *t-tupla* na solução final. A Tabela 3.5 mostra Θ para o exemplo apresentado na Tabela 3.4 e $t = 2$. Note que as interações são feitas para os níveis dos fatores $banco \backslash funcao$, $banco \backslash bandeira$ e $funcao \backslash bandeira$. Inicialmente todos os valores de *flag* são iguais a *falso*. O Algoritmo 2 mostra o procedimento *Construtor* onde $*$ é um operador de concatenação e a Figura 3.3 descreve o procedimento na forma de fluxograma.

Figura 3.3 - Fluxograma do procedimento Construtor 1.0



O *Construtor* opera da seguinte maneira: com base no conjunto de fatores (domínio), f, e no valor de t, as interações entre os fatores são geradas por meio do procedimento enumerador, e armazenadas em um conjunto chamado E (linha 1). Por exemplo, tem-se 3 fatores ($bandeira$, $banco$ e $funcao$) e $t = 2$, sabemos que o enumerador vai gerar as interações 2 a 2 (por causa do valor de t) entre esses 3 fatores: $banco \backslash funcao$, $banco \backslash bandeira$ e $funcao \backslash bandeira$. Em seguida, as interações (I_i) são selecionadas

[5] Θ é uma matriz cuja ordem varia. Em termos de implementação do algoritmo TTR, embora o número de colunas seja conhecido de antemão (f), o número de linhas ($|C|$) depende das combinações *t-way* dos níveis dos fatores. Durante o processo de realocação, o TTR remove as linhas até que Θ esteja vazia.

Tabela 3.5 - Matriz Θ para o exemplo da Figura 3.2

i	**banco**	**função**	**bandeira**	**flag**
1	*BancoA*	*debito*		$falso$
2	*BancoA*	*credito*		$falso$
3	*BancoB*	*debito*		$falso$
4	*BancoB*	*credito*		$falso$
5	*BancoA*		*CartaoA*	$falso$
6	*BancoA*		*CartaoB*	$falso$
7	*BancoA*		*CartaoC*	$falso$
8	*BancoB*		*CartaoA*	$falso$
9	*BancoB*		*CartaoB*	$falso$
10	*BancoB*		*CartaoC*	$falso$
11		*debito*	*CartaoA*	$falso$
12		*debito*	*CartaoB*	$falso$
13		*debito*	*CartaoC*	$falso$
14		*credito*	*CartaoA*	$falso$
15		*credito*	*CartaoB*	$falso$
16		*credito*	*CartaoC*	$falso$

uma por vez (linha 3), e durante essa seleção, as *t-tuplas* são construídas com base em cada fator dessa interação: na linha 4, o primeiro fator da primeira interação é selecionado, e cada nível (l_i) é transformado em uma *t-tupla* (θ_i) (linha 6) e são incluídas em Θ (linha 7). A partir daí, os fatores subsequentes são selecionados um a um, e uma nova *t-tupla* é gerada a partir da combinação entre cada nível (l_j) com cada *t-tupla* (θ_i) pré-existente em Θ (linha 12). Por exemplo, o algoritmo vai selecionar a primeira interação gerada, $banco \backslash funcao$, e construir todas as *t-tuplas* entre esses dois fatores.

3.1.2 Solução Inicial

A matriz $M_{|n|\times|(f+1)|}$ é o MCNV que irá conter o conjunto de casos de teste, onde $|n|$ são as combinações de fatores (casos de teste), $|f|$ a quantidade de fatores e a coluna adicional ($(f + 1)$-ésima coluna) armazenará o parâmetro ζ, que será melhor explicado adiante. Existe uma solução inicial para a matriz M que é obtida pela seleção da interação de fatores I_i que tenha a maior quantidade de *t-tuplas* ainda não-cobertas. Considerando a ordem de entrada *banco*, $funcao$ e *bandeira*, $I_2 = \{banco, bandeira\}$ é escolhida porque ela possuí 6 *t-tuplas* e está armazenada antes que $I_3 = \{funcao, bandeira\}$ (repare nos indices da Tabela 3.5). Todas as *t-tuplas* derivadas via I_2 consideradas na solução inicial são combinadas a casos de teste vazios, respeitando a ordem de entrada dos fatores/níveis para

Algorithm 2 O procedimento Construtor: TTR 1.0

entrada: $f = \{f_i \mid i = 1...|f|\}, t$
saída: $\Theta_{|C|\times|f|} = \{\theta_i \mid i = 1...m\}$

```
1:  E ← enumerador(f, t)
2:  while E ≠ ∅ do
3:      let I_i ⊂ E
4:      let F_1 ⊂ I_i
5:      for all {l_i} ∈ F_1 do
6:          θ_i ← θ_i ∪ {l_i}
7:          Θ ← Θ * θ_i
8:      while |F_j| ≤ t, |F_j| ∈ I_i, j > 1 do
9:          for all {l_j} ∈ F_j do
10:             for all θ_i ⊂ Θ do
11:                 Θ ← Θ * (θ_i ∪ {l_j})
12:     E ← E \ I_i
```

o TTR 1.0 como ilustrado na matriz (a) da Figura 3.5 (repare nas *t-tuplas* $\theta_5 = \{BancoA, CartaoA\}$), $\theta_6 = \{BancoA, CartaoB\}$, ... de Θ (Tabela 3.5) em M).

3.1.3 Metas

O critério para a mudança da solução corrente é o parâmetro *meta* (ζ). Para cada linha da matriz, ou seja, para cada caso de teste, existe um parâmetro *meta* associado.

Considerando que o objetivo é cobrir o maior número de *t-tuplas* não-cobertas, o parâmetro *meta* é calculado de acordo com o máximo número de *t-tuplas* ainda não-cobertas $(\theta_i, \theta_j, ...)$ que potencialmente podem ser cobertas quando a *t-tupla* θ_k é movida de Θ para M. Isso resulta em um caso de teste temporário τ. A fim de encontrar o valor de *meta*, é necessário levar em conta o seguinte: (i) os fatores disjuntos cobertos pela *t-tupla* representada por f_d; (ii) a quantidade de *t-tuplas* que foram determinadas por essa combinação e que já foram cobertas, representada por y; e (iii) o *strength* t.

$$\zeta = \binom{f_d}{t} - y.$$

Ainda considerando a Tabela 3.4 e $t = 2$, de acordo com a matriz Θ (veja a Tabela 3.5), a solução inicial M (Figura 3.5), é composta pelas *t-tuplas* da interação de

fatores $banco \backslash bandeira$. Isto porque a interação de fatores $banco \backslash funcao$ tem 4 *t-tuplas*, $banco \backslash bandeira$ tem 6 *t-tuplas*, e $funcao \backslash bandeira$ também tem 6 *t-tuplas* como é possível constatar na matriz Θ. Como $banco \backslash bandeira$ vem antes que $funcao \backslash bandeira$ e ambos possuem 6 *t-tuplas*, o algoritmo selecioná-la para ser realocada em M.

A quantidade de fatores disjuntos, f_d, é igual a 3. Como a interação $banco \backslash bandeira$ já foi contemplada na matriz M, a próxima interação de fatores que prove o maior número de *t-tuplas* não-cobertas é $funcao \backslash bandeira$. Depois, tem-se todos os 3 fatores com $banco \backslash bandeira$ e $funcao \backslash bandeira$, o que explica $f_d = 3$. Como $t = 2$, tem-se $\binom{3}{2} = 3$. Contudo, uma das 3 interações de fatores já foi coberta durante a construção da solução inicial ($banco \backslash bandeira$), então é necessário cobrir apenas 2 interações de fatores. Assim, $y = 1$. Portanto, para cada *t-tupla* na solução inicial M ainda falta cobrir:

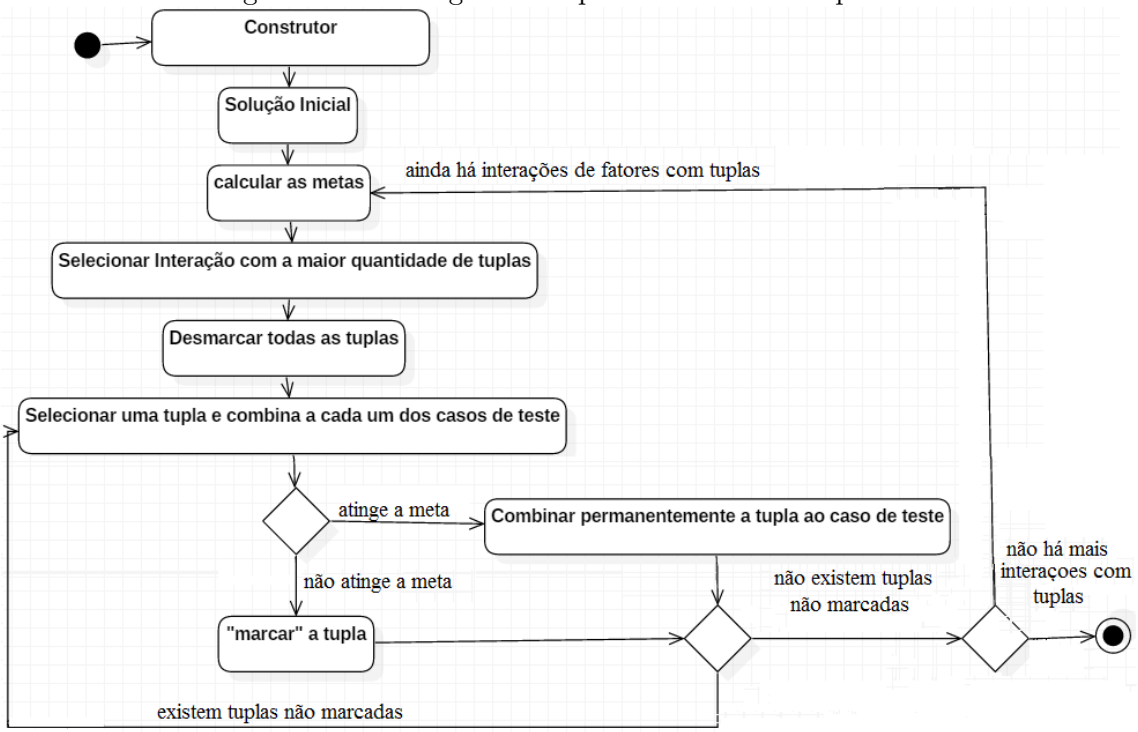
$$\zeta = \binom{3}{2} - 1 = 2.$$

Isso explica o parâmetro meta (ζ) na Figura 3.5. É muito importante que y seja subtraído a fim de se achar ζ. Se isso não for feito, a meta final nunca vai ser encontrada, uma vez que não existem *t-tuplas* ainda não-cobertas correspondentes a essa interação.

3.1.4 O Procedimento *Principal*

O Algoritmo 3 apresenta o procedimento *Principal* do TTR 1.0 e a Figura 3.4 mostra o procedimento descrito na forma de fluxograma. Depois da construção da matriz Θ (processo: *Construtor*), a solução inicial, e o cálculo das metas de todas as *t-tuplas*, o procedimento *Principal* seleciona a interação de fatores que possui a maior quantidade de *t-tuplas* ainda não-cobertas (linha 2 do Algoritmo). Contudo, essas tuplas não vão ser realocadas de Θ para a matriz M de uma única vez. Elas vão ser realocadas gradativamente, uma por uma, à medida que as metas sejam atingidas (linha 5 e linha 8 do Algoritmo).

Figura 3.4 - Fluxograma do procedimento Principal 1.0



A Figura 3.5 ajuda a entender o conceito do procedimento *Principal*. Todas as matrizes na figura representam *snapshots* da matriz M. A matriz superior a esquerda (a) é a solução inicial. Enquanto existirem *t-tuplas* em Θ, o procedimento *Principal* prossegue. Assim, o algoritmo *Principal* seleciona de Θ a maior quantidade de *t-tuplas* ainda não cobertas. No exemplo da Tabela 3.4, as *t-tuplas* foram selecionadas da interação de fatores $funcao \backslash bandeira$. Toda *t-tupla* da interação $funcao \backslash bandeira$ foi comparada com a matriz M.

Quando uma *t-tupla* não coberta se encaixa em uma linha já existente em M, a fim de completar o caso de teste, i.e., a *t-tupla* faz com que aquela linha de M fique com todos os níveis do cojunto de fatores, significa que a *meta* para aquela linha em M foi atingida. Vamos considerar a primeira linha da matriz M, que é um caso de teste originado das *t-tuplas* θ_5 e θ_{11}. A inserção de θ_{11} em M é aceita porque a meta $\zeta = 2$ é atingida. Em outras palavras, pela inserção de θ_{11}, nós temos um caso de teste completo $\tau = \{BancoA, debito, CartaoA\}$. Dessa maneira, as outras duas interações de fatores ($I_1 = \{banco, funcao\} \rightarrow \theta_1; I_3 = \{funcao, bandeira\} \rightarrow \theta_{11}$) são cobertas, e a meta é atingida. A matriz superior à direita (b) na Figura 3.5

Algorithm 3 O procedimento Principal: TTR 1.0

entrada: $\Theta_{|C|\times|f|} = \{\theta_i \mid i = 1...m\}$
saída: $M_{|n|\times|(f+1)|} = \{\tau_i \mid i = 1..|n|\}$

1: **for all** $\theta_i \subset \Theta$ **do**
2: **while** $\theta_i \not\subset M$ **do**
3: **for all** $\tau_i \subset M$ **do**
4: $\tau_i \leftarrow \tau_i \cup \theta_i$
5: **if** $|\tau_i| = t$ **then**
6: **if** $goal(\tau_i)$ **then** (verifica se o caso de teste τ_i atinge a meta)
7: $\Theta \leftarrow \Theta \setminus \theta_i$
8: **else**
9: $\tau_i \leftarrow \tau_i \setminus \theta_i$
10: $marcar(\theta_i)$ ($flag$ assume o valor $verdadeiro$)

mostra o resultado dessa primeira inserção.

Figura 3.5 - Solução Final M: cojunto de casos de teste

(a)

banco	função	bandeira	ζ
BancoA		*CartaoA*	2
BancoA		*CartaoB*	2
BancoA		*CartaoC*	2
BancoB		*CartaoA*	2
BancoB		*CartaoB*	2
BancoB		*CartaoC*	2

(b)

banco	função	bandeira	ζ
BancoA	*debito*	*CartaoA*	0
BancoA	*credito*	*CartaoB*	0
BancoA		*CartaoC*	2
BancoB	*credito*	*CartaoA*	0
BancoB	*debito*	*CartaoB*	0
BancoB		*CartaoC*	2

(c)

banco	função	bandeira	ζ
BancoA	*debito*	*CartaoA*	0
BancoA	*credito*	*CartaoB*	0
BancoA		*CartaoC*	1
BancoB	*credito*	*CartaoA*	0
BancoB	*debito*	*CartaoB*	0
BancoB		*CartaoC*	1

(d)

banco	função	bandeira	ζ
BancoA	*debito*	*CartaoA*	0
BancoA	*credito*	*CartaoB*	0
BancoA	*debito*	*CartaoC*	0
BancoB	*credito*	*CartaoA*	0
BancoB	*debito*	*CartaoB*	0
BancoB	*credito*	*CartaoC*	0

Depois da atualização, o novo valor de ζ é calculado. A matriz inferior a esquerda (c) mostra os novos valores de ζ (veja as linhas 3 e 6). Assim os passos descritos acima são repetidos com a inserção/realocação de *t-tuplas* na matriz M. Uma vez

que uma *t-tupla* não-coberta de Θ é incluída na matriz M, essa *t-tupla* é excluída de Θ assim como todas as outras *t-tuplas* de Θ cobertas por esse caso de teste (completo). O conjunto final de casos de teste é a matriz M ilustrada à direita inferior (d) da Figura 3.5.

Existe ainda a possibilidade que uma certa *t-tupla* ainda não-coberta não se encaixe em M. Consequentemente, o parâmetro *flag* dessa *t-tupla* em Θ é marcado como *verdadeiro* para que o procedimento *Principal* entenda que essa *t-tupla* não pode mais ser comparada com as linhas de M. O procedimento *Principal* continua processando enquanto houver *t-tuplas* não-cobertas. A Figura 3.6 mostra a matriz Θ depois da primeira iteração do exemplo. Note que, até o momento, as *t-tuplas* $\theta_{13} = \{debito, CartaoC\}$ e $\theta_{16} = \{credito, CartaoC\}$ da interação de fatores $funcao \backslash bandeira$ ainda não se encaixaram na matriz M (veja os valores *verdadeiro*).

Tabela 3.6 - Matriz M para o exemplo da Figura 3.2.

i	**banco**	**função**	**bandeira**	**flag**
13		*debito*	*CartaoC*	*verdadeiro*
16		*credito*	*CartaoC*	*verdadeiro*

Essa exceção com as *t-tuplas* $\theta_{13} = \{debito, CartaoC\}$ e $\theta_{16} = \{credito, CartaoC\}$ acontece porque os testes gerados por essas *t-tuplas* e as linhas disponíveis na matriz M cobrem *t-tuplas* já cobertas na matriz Θ. Considerando que se o teste consistir da combinação da *t-tupla* $\theta_{13} = \{debito, CartaoC\}$ e a linha 3 da matriz M, somente uma *t-tupla* ainda não-coberta vai ser coberta (no caso, ela própria), uma vez que não existe mais *t-tuplas* a serem cobertas nas interações $banco \backslash bandeira$ e $banco \backslash funcao$, como ilustrado na Figura 3.6. Contudo, a *meta* para aquela linha ($\zeta = 2$) não foi atingida e essas *t-tuplas* não podem ser removidas da matriz Θ. Então é necessário calcular novamente as *metas* de acordo com as interações de fatores já cobertas.

3.2 TTR: Versão 1.1

Conforme comentado na seção anterior, uma característica da versão 1.0 do algoritmo TTR é a insensibilidade à ordem em que fatores e níveis são inseridos no algoritmo. Essa característica é devida à ordenação dos fatores antes da execução do algoritmo. Dessa maneira, independentemente da ordem, os conjuntos de teste

possuem sempre a mesma quantidade de casos de teste, e preservam os casos de teste com as mesmas interações de fatores/níveis. A versão 1.1 do algoritmo proposto não realiza esse processo de ordenação. Dessa maneira, o novo algoritmo pode produzir conjuntos de teste com tamanhos diferentes, assim como os outros algoritmos existentes na literatura. A motivação para esse fato é que foi percebido uma melhora, embora não em todas as instâncias, do fato de não ordenar (versão 1.1) se comparado à ordenação (versão 1.0). Com essa pequena mudança, tornamos os resultados mais flexíveis, uma vez que podemos ter várias opções de conjunto de teste apenas variando a ordem de entrada dos fatores. Isso significa que diferentes conjuntos de teste que resolvem o mesmo problema, tem diferentes potenciais para a revelação de defeitos.

Algorithm 4 O procedimento Construtor: TTR 1.1

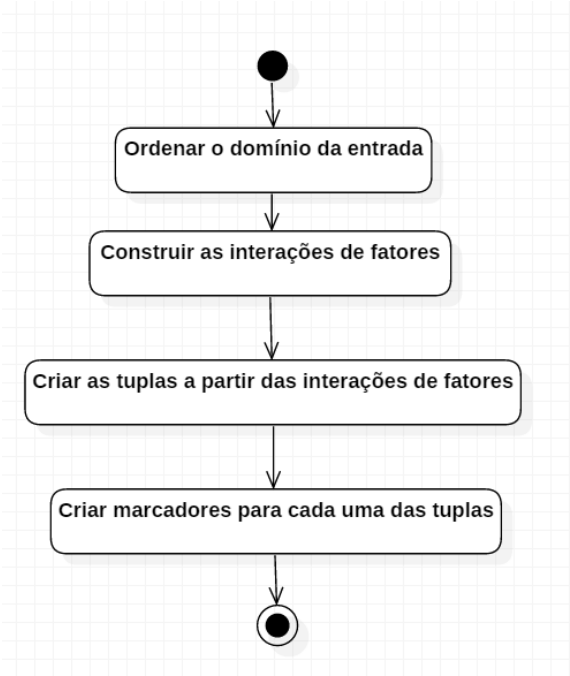
entrada: $f = \{f_i \mid i = 1...|f|\}, t$
saída: $\Theta_{|C|\times|f|} = \{\theta_i \mid i = 1...m\}$

1: $E \leftarrow enumerador(f, t)$
2: **while** $E \neq \emptyset$ **do**
3: **let** $I_i \subset E$
4: **let** $F_1 \subset I_i$
5: **for all** $\{l_i\} \in F_1$ **do**
6: $\theta_i \leftarrow \theta_i \cup \{l_i\}$
7: $\Theta \leftarrow \Theta * \theta_i$
8: **while** $|F_j| \leq t, |F_j| \in I_i, j > 1$ **do**
9: **for all** $\{l_j\} \in F_j$ **do**
10: **for all** $\theta_i \in \Theta$ **do**
11: $\Theta \leftarrow \Theta * (\theta_i \cup \{l_j\})$
12: $E \leftarrow E \setminus I_i$

Os Algoritmos mostrados nas Figuras 4 e 5 mostram os algoritmos da versão 1.1 do TTR e a Figura 3.6 o procedimento Construtor descrito na forma de fluxograma. Nota-se que eles diferem dos Algoritmos da versão 1.0 (Figura 2 e Figura 3) em apenas dois pontos: (i) a linha 1 do Algoritmo 4 mostra a ordenação do dominio, ou seja, os fatores são ordenados de acordo com a quantidade de níveis que eles possuem - da maior quantidade para a menor quantidade. Por exemplo, tomando o funcionamento do algoritmo 2 e considerando os fatores e níveis da Tabela 3.4, que foram passados na seguinte ordem: *banco* (com os níveis: $BancoA$ e $BancoB$), $funcao$ (com os níveis: *debito* e *credito*) e por fim *bandeira* (com os níveis: $CartaoA$, $CartaoB$ e $CartaoC$), dão origem à matriz Θ ilustrada na Tabela 3.5. Quando a

solução inicial é construída (Matriz M), é necessário que o algoritmo ache a interação de fatores com a maior quantidade de *t-tuplas*, então é necessário que o algoritmo percorra a matriz Θ por completo.

Figura 3.6 - Fluxograma do procedimento Construtor 1.1



Algorithm 5 O procedimento Principal: TTR 1.1

entrada: $\Theta_{|C|\times|f|} = \{\theta_i \mid i = 1...m\}$
saída: $M_{|n|\times|(f+1)|} = \{\tau_i \ |i = 1..|n|\}$

for all $\theta_i \subset \Theta$ **do**
  while $\theta_i \not\subset M$ **do**
    for all $\tau_i \subset M$ **do**
      $\tau_i \leftarrow \tau_i \cup \theta_i$
      if $|\tau_i| = t$ **then**
        if $goal(\tau_i)$ **then** (verifica se o caso de teste τ_i atinge a meta)
          $\Theta \leftarrow \Theta \setminus \theta_i$
        else
          $\tau_i \leftarrow \tau_i \setminus \theta_i$
    $marcar(\theta_i)$ ($flag$ assume $verdadeiro$)

Da perspectiva da versão 1.1, embora os fatores fossem passados nessa mesma ordem, eles seriam armazenados de forma ordenada, como mostra a Tabela 3.7: o primeiro

fator passa a ser *bandeira* (que possui 3 níveis, a maior quantidade em relação aos outros fatores), o segundo fator passa a ser *banco* (que possui 2 níveis) e o terceiro fator passa a ser $funcao$ (que possui 2 níveis). Esse domínio dá origem à matriz Θ ilustrada pela Tabela 3.8, em que de fato, as interações de fatores com maior quantidade de *t-tuplas* estão nas primeiras posições, dessa maneira, quando a solução inicial for construída, basta buscar a primeira interação de fatores da matriz Θ, não sendo mais necessário percorre-la por inteira. Além disso, esse procedimento, de busca por interações de fatores, não é feito somente para a construção da solução inicial, ele é feito em mais pontos no procedimento *Principal* (linha 2 da Figura 5), durante a construção da solução final (matriz M).

Tabela 3.7 - Exemplo de fatores e níveis

Fator	**Nível**
bandeira	$CartaoA$, $CartaoB$, $CartaoC$
banco	$BancoA$, $BancoB$
funcao	*debito*, *credito*

Tabela 3.8 - Matriz Θ construída a partir do domínio ordenado, para o exemplo da Figura 3.2.

i	**bandeira**	**banco**	**funcao**	**flag**
1	$CartaoA$	$BancoA$		$falso$
2	$CartaoA$	$BancoB$		$falso$
3	$CartaoB$	$BancoA$		$falso$
4	$CartaoB$	$BancoB$		$falso$
5	$CartaoC$	$BancoA$		$falso$
6	$CartaoC$	$BancoB$		$falso$
7	$CartaoA$		*debito*	$falso$
8	$CartaoA$		*credito*	$falso$
9	$CartaoB$		*debito*	$falso$
10	$CartaoB$		*credito*	$falso$
11	$CartaoC$		*debito*	$falso$
12	$CartaoC$		*credito*	$falso$
13		$BancoA$	*debito*	$falso$
14		$BancoA$	*credito*	$falso$
15		$BancoB$	*debito*	$falso$
16		$BancoB$	*credito*	$falso$

3.3 TTR: Versão 1.2

O Algoritmo geral do TTR versão 1.2 é ilustrado na Figura 6. Em contraste com o algoritmo geral das versões anteriores, não faz mais uso do procedimento *Construtor* (ver Versão 1.0, Figura 1), uma vez que as *t-tuplas* são geradas uma por vez, à medida em que são realocadas. Em outras palavras, não existe mais matriz Θ. Agora, existe apenas a matriz de interações de fatores φ. TTR 1.2 opera da seguinte forma: (i) gera somente as interações de fatores (ou seja, ainda não gera nenhuma *t-tupla*); (ii) gera a solução inicial e armazena em M; e (iii) as *t-tuplas* são geradas a partir de φ a medida em que são inseridas em M, por meio do procedimento *Principal*.

Algorithm 6 Visão geral: TTR 1.2

entrada: $f = \{f_i \mid i = 1...|f|\}, t$
saída: $M_{|n|\times|(f+1)|} = \{\tau_i \mid i = 1..|n|\}$

1: **while** $\exists\ \tau \mid \tau \in M \wedge\ \zeta > 0$ **do**
2: $M \leftarrow calcularSolucaoInicial(\varphi)$
3: $[M, \varphi] \leftarrow Principal(M, \varphi, \zeta)$
4: $\zeta \leftarrow calcularZeta(M)$

Considere o exemplo apresentado na Figura 3.2, onde os níveis e fatores estão apresentados na Tabela 3.9 e $t = 3$.

Exemplo 3.2: Uma escola que oferece cursos técnicos possui um sistemas de cadastro de professores, em que um dos seus requisitos funcionais é *atualizar informações gerais*. Esse requisito é implementado por meio do método *atualizarInformaçõesGerais* (ver Figura 3.7), que recebe informações relacionadas ao *status* (se está aposentado ou não), modalidade de ensino (se o professor ministra aulas para cursos técnicos ou somente para o ensimo medio), regime de atuação (parcial ou integral) e turno (vespertino e matutino).

Figura 3.7 - Assinatura do método *atualizarInformaçõesGerais*.

```
public void atualizarInformaçõesGerais(Status status, Modalidade modalidade, Regime regime, Turno turno){}
```

Os níveis e fatores do Exemplo 3.2 estão relacionados na Tabela 3.9. Ao contrá-

Tabela 3.9 - Exemplo de fatores e níveis.

Fator	**Nível**
status	*ativo*, *aposentado*
modalidade	*medio*, *tecnico*
regime	*parcial*, *integral*
turno	*vespertino*, *matutino*

rio das versões anteriores, o algoritmo TTR versão 1.2 constrói apenas as interações de fatores de acordo com o valor de t e armazena o número correspondente de *t-tuplas* em φ. Com base no Exemplo 3.2, as interações de fatores construídas serão: $I_1 = \{status, modalidade, regime\}$, $I_2 = \{status, modalidade, turno\}$, $I_3 = \{status, regime, turno\}$ e $I_4 = \{modalidade, regime, turno\}$. Nenhuma *t-tupla* correspondente a nenhuma das interações de fatores e níveis será construída, como mostra a Tabela 3.11 (repare que os "x" servem para marcar o fator que faz parte daquela interação). A cada uma dessas interações será associado um parâmetro, Φ, que faz parte de φ, e irá armazenar a quantidade de *t-tuplas* correspondentes a cada uma dessas interações. Esse cálculo é feito através da multiplicação da quantidade de níveis de cada fator existente na interação. Por exemplo, a interação $I_1 = \{status, modalidade, regime\}$ possui três fatores com respectivamente 2, 2 e 2 níveis cada, portanto, a quantidade de *t-tuplas* que pertencerá a ela é $2 * 2 * 2 = 8$, como mostra a Tabela 3.10.

Tabela 3.10 - Quantidade de *t-tuplas* prevista para cada uma das interações de fatores.

Interação	Φ
$(status, modalidade, regime)$	2 * 2 * 2 = 8
$(status, modalidade, turno)$	2 * 2 * 2 = 8
$(status, regime, turno)$	2 * 2 * 2 = 8
$(modalidade, regime, turno)$	2 * 2 * 2 = 8

Tabela 3.11 - Matriz Θ para o exemplo da Figura 3.7.

Interação de Fatores	**status**	**modalidade**	**regime**	**turno**	Φ
I_1	x	x	x		8
I_2	x	x		x	8
I_3	x		x	x	8
I_4		x	x	x	8

3.3.1 Solução Inicial

Tanto a solução inicial quanto a solução final são armazenadas na matriz M. No fundo, a solução inicial nada mais é do que a construção das *t-tuplas* da interação de fatores com maior valor de Φ e sua transformação em casos de teste. Observe a matriz superior a esquerda (a) na Figura 3.8: as *t-tuplas* da interação de fatores $I_1 = \{status, modalidade, regime\}$ foram todas transformadas em casos de teste. Portanto, para essa interação de fatores o valor de Φ passa a ser 0, e ela não é mais considerada no cálculo das metas (Tabela 3.12). De fato, estamos considerando um exemplo com 4 fatores e $t = 3$, portanto são geradas 4 interações de fatores possíveis; uma já está coberta, dessa forma, restam 3 interações de fatores (I_2, I_3 e I_4) ainda não cobertas como mostra a Tabela 3.12, isso justifica o valor de ζ.

Figura 3.8 - Solução Final M: cojunto de casos de teste

(a)

status	modalidade	regime	turno	ζ
ativo	*medio*	*parcial*		3
ativo	*tecnico*	*parcial*		3
aposentado	*medio*	*parcial*		3
aposentado	*tecnico*	*parcial*		3
ativo	*medio*	*integral*		3
ativo	*tecnico*	*integral*		3
aposentado	*medio*	*integral*		3
aposentado	*tecnico*	*integral*		3

(b)

status	modalidade	regime	turno	ζ
ativo	*medio*	*parcial*	*vespertino*	0
ativo	*tecnico*	*parcial*		3
aposentado	*medio*	*parcial*		3
aposentado	*tecnico*	*parcial*	*vespertino*	0
ativo	*medio*	*integral*		3
ativo	*tecnico*	*integral*	*vespertino*	0
aposentado	*medio*	*integral*	*vespertino*	0
aposentado	*tecnico*	*integral*		3

(c)

status	modalidade	regime	turno	ζ
ativo	*medio*	*parcial*	*vespertino*	0
ativo	*tecnico*	*parcial*	*matutino*	0
aposentado	*medio*	*parcial*	*matutino*	0
aposentado	*tecnico*	*parcial*	*vespertino*	0
ativo	*medio*	*integral*	*matutino*	0
ativo	*tecnico*	*integral*	*vespertino*	0
aposentado	*medio*	*integral*	*vespertino*	0
aposentado	*tecnico*	*integral*	*matutino*	0

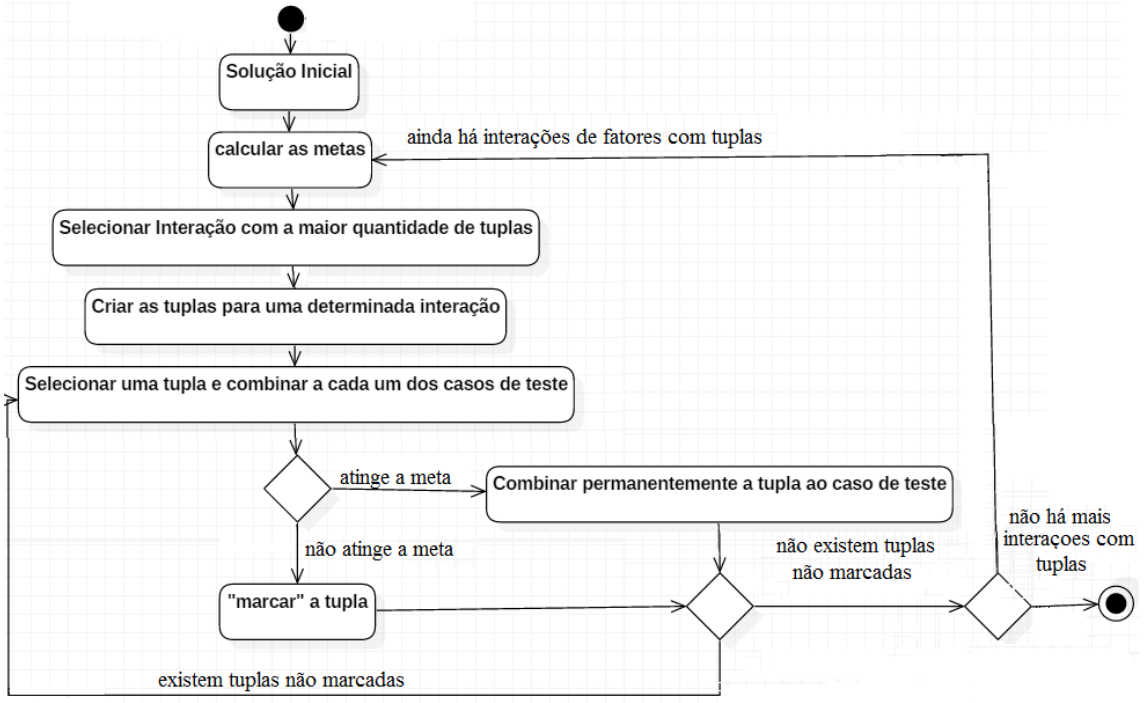
Tabela 3.12 - Matriz Θ para o exemplo da Figura 3.7

Interação de Fatores	**status**	**modalidade**	**regime**	**turno**	Φ
I_2	x	x		x	8
I_3	x		x	x	8
I_4		x	x	x	8

3.3.2 O Procedimento *Principal*

O procedimento *Principal* toma a solução inicial para chegar à solução final. O Algoritmo 7 apresenta o procedimento *Principal* do TTR, versão 1.2 e a Figura 3.9 o procedimento descrito na forma de fluxograma. Depois do cálculo das interações de fatores e dos valores de Φ para cada uma delas, a solução inicial, e o cálculo das *metas* de todos os casos de teste da matriz M, o procedimento *Principal* seleciona a interação de fatores que possui a maior quantidade de *t-tuplas* ainda não-cobertas (linha 2), e constrói as *t-tuplas*, para que elas possam ser realocadas, uma a uma. Contudo, essas *t-tuplas* não vão ser realocadas de uma única vez. Elas vão ser realocadas gradativamente, uma por uma, à medida que as metas sejam atingidas (linha 15).

Figura 3.9 - Fluxograma do procedimento Principal 1.2



Tomando o Exemplo 3.2, a interação de fatores com a maior quantidade de *t-tuplas* ainda não cobertas é $I_2 = \{status, modalidade, turno\}$, com $\Phi = 8$ (Tabela 3.12): todas as *t-tuplas* dessa interação são geradas e armazenadas em S (linha 3). A primeira *t-tupla*, $s_1 = \{ativo, medio, vespertino\}$ é comparada a cada um dos casos de teste de M, sendo verificado se a mesma se encaixa em al-

Algorithm 7 O procedimento Principal: TTR 1.2

entrada: $\varphi_{|I|\times|(f+1)|} = \{I_i \mid i = 1...m\}$
saída: $M_{|n|\times|(f+1)|} = \{\tau_i \mid i = 1..n\}$

1: **for all** $I_i \subset \varphi$ **do**
2: **let** $\phi_i \subset I_i$ (Seleciona a interação de fatores que tem a maior quantidade de t-tuplas que ainda não foram cobertas (i.e. o maior valor de Φ))
3: $S \leftarrow buildTuples(\phi_i)$
4: **let** $\theta_i \subset S$ (θ_i é uma t-tupla)
5: **while** $\phi_i \neq \emptyset$ **do**
6: **for all** $\tau_i \subset M$ **do**
7: $\tau_i \leftarrow \tau_i \cup \theta_i$
8: **if** $|\tau_l| \geq t$ **then**
9: **if** $goal(\tau_i)$ **then** (Verifica se o caso de teste τ_i possuí ζ t-tuplas inéditas)
10: $S \leftarrow S \setminus \theta_i$
11: **for all** $I_j \subset \varphi \wedge I_j \subset \tau_i$ **do**
12: **let** $\{\Phi\} \in I_j$
13: $\{\Phi\} \leftarrow \{\Phi\}$ - 1
14: **else**
15: $\tau_i \leftarrow \tau_i \setminus \theta_i$
16: $marcar(\theta_i)$ ($flag$ assume $verdadeiro$)
17: $\varphi \leftarrow \varphi \setminus I_i$

gum caso de teste τ (linha 7). A tupla em questão irá se encaixar em τ_1, então é gerado um caso de teste auxiliar, composto pelos valores da *t-tupla* e de τ_1 (linha 7). Esse caso de teste então é "quebrado" em *t-tuplas* (linha 9), de acordo com as interações de fatores que possuem Φ diferente de 0. Por exemplo, o teste auxiliar é $\tau_1 = \{ativo, medio, parcial, vespertino\}$, é quebrado nas *t-tuplas*: $\{\{ativo, medio, parcial\}, \{ativo, medio, vespertino\}, \{ativo, parcial, vespertino\}, \{medio, parcial, vespertino\}\}$. É verificado então quantas dessas *t-tuplas* são inéditas, ou seja, que ainda não existem na matriz M, e se a quantidade contabilizada for igual ao valor de ζ_1, então o τ_1 é alocado permanentemente na matriz M. Uma vez que a matriz é percorrida por um laço (linha 6), ela será atualizada toda vez em que uma *t-tupla* for combinada com algum caso de teste (linha 7).

Esse passo é repetido para todas as *t-tuplas*. Cada vez que uma *t-tupla* é realocada de S em M, as *metas* são calculadas novamente. Por exemplo, quando a matriz M recebe permanentemente a 4ª *t-tupla*, os casos de teste que se tornam completos (com um nível para cada fator) passam a ter o valor de $\zeta = 0$, enquanto os demais possuem ainda $\zeta = 3$ (matriz superior a direita (b) da Figura 3.8).

Todas as *t-tuplas* de I_2 são realocadas de S de maneira a atingir a *meta* de todos os casos de teste de M, resultando na matriz inferior (c) da Figura 3.8. De fato, o algoritmo *Principal* não constrói novas *t-tuplas* originadas de outra interação de fatores se a atual não tiver sido zerada, i.e., se a interação de fatores I_2, que foi selecionada como a interação com o maior valor de Φ, ainda possuir *t-tuplas*, o algoritmo não irá selecionar outra interação de fatores para utilizar novas *t-tuplas* para atingir a *meta* dos casos de teste de M. Para isso, a *meta* dos casos de teste será diminuída, unidade por unidade, até que todas as *t-tuplas* da interação de fatores I_2 façam os casos de teste atingirem suas *metas*. No entanto, existe a exceção de que alguma *t-tupla* possa vir a não se encaixar em nenhum caso de teste existente em M. Nessa situação, a *t-tupla* é transformada em um novo caso de teste, e adicionada em M.

3.4 Considerações Finais Sobre esse Capítulo

Esse capítulo apresentou o algoritmo TTR na versão 1.0 (BALERA; SANTIAGO JÚNIOR, 2015), na versão 1.1 (BALERA; SANTIAGO JÚNIOR, 2016), em que se fez uma mudança relacionada à não ordenação dos fatores de entrada: o algoritmo gera conjuntos de teste independentes da ordem de entrada dos fatores, o que soluciona o problema do algoritmo produzir diferentes conjuntos de teste dependendo da ordem que se apresenta os fatores, e propôs a sua última versão, 1.2, em que o algoritmo constrói o conjunto de teste sem precisar partir da solução mais custosa, ou seja, não gera a matriz com todas as *t-tuplas* a serem cobertas.

A avaliação experimental relacionada ao algoritmo TTR é apresentada no próximo capítulo.

4 AVALIAÇÃO EXPERIMENTAL

Esse capítulo apresenta a avaliação experimental realizada para determinar se o algoritmo TTR possui melhor, ou ao menos desempenho similar, a outros algoritmos/ferramentas da literatura que geram *designs* combinatoriais, particularmente MCNVs. Os algoritmos/ferramentas aos quais o TTR foi comparado foram o IPOG-F (FORBES et al., 2008), *jenny* (JENKINS, 2016), IPO-TConfig (WILLIAMS, 2000) e PICT (CZERWONKA, 2006). De fato, a avaliação experimental que será apresentada é composta por três avaliações rigorosas: dois experimentos controlados e um quasiexperimento (WOHLIN et al., 2000; KUHN et al., 2004; ZANNIER et al., 2006; LEMOS et al., 2013). É importante destacar que, o primeiro experimento controlado envolve a versão 1.1 do TTR, e o segundo experimento controlado e o quasiexperimento envolvem a versão 1.2 do algoritmo TTR.

4.1 Experimento Controlado 1: Custo e Similaridade

4.1.1 Definição e Contexto

O objetivo principal deste estudo é avaliar duas perspectivas de custo relacionadas à geração de casos de teste por meio de *designs* combinatoriais considerando nossa solução, versão 1.1 do algoritmo TTR, e quatro outros algoritmos/ferramentas propostos na literatura: IPOG-F (FORBES et al., 2008), *jenny* (JENKINS, 2016), IPO-TConfig (WILLIAMS, 2000) e PICT (CZERWONKA, 2006). A primeira definição de custo refere-se ao tamanho dos conjuntos de teste. A segunda definição de custo refere-se ao tempo para gerar os conjuntos de testes, com base em cada algoritmo/ferramenta. Devemos enfatizar que esse tempo não é o tempo para executar os conjuntos de testes derivados de cada algoritmo. Uma terceira comparação foi feita, onde a similaridade entre os conjuntos de testes geradas foi analisada. Aqui, o objetivo é perceber se os casos de teste, gerados por uma determinada abordagem, diferem ou se assemelham aos casos de teste gerados por outra solução.

O experimento foi conduzido pelos pesquisadores que o definiram. O processo de experimentação proposto em (WOHLIN et al., 2000) foi utilizado como base para a realização deste experimento controlado, utilizando-se a ferramenta R (KOHL, 2015) para análise de custos e python (MATTHES, 2016) para a análise de similaridade.

Como o objetivo é basicamente comparar o custo de várias soluções na literatura com a solução proposta nesse trabalho, o conjunto de amostras é, de fato, formado por instâncias que serão submetidas aos algoritmos/ferramentas para a geração das

matrizes (conjuntos de testes). Inicialmente, foram escolhidas 33 instâncias/amostras de teste (compostas de fatores e níveis) com a *strength*, t, variando de 2 a 6. No entanto, apenas 27 das 33 instâncias puderam gerar casos de teste para todos os algoritmos: em 6 instâncias, IPO-TConfig falhou em terminar sua execução (a janela do IPO-Tconfig fechou sozinha, impedindo que a execução continuasse), mesmo após 24 horas de execução e, portanto, foram descartadas essas 6 instâncias. A Tabela 4.1 mostra as 27 instâncias/amostras utilizadas neste estudo.

É importante caracterizar cada instância/amostra. Consideremos a instância $i = 1$ na Tabela 4.1:

$$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1, \quad strength = 2 \tag{4.1}$$

Por exemplo, no contexto da geração de casos de teste unitários para programas desenvolvidos de acordo com o paradigma de Programação Orientada a Objetos (PPO), esta instância pode ser usada para gerar casos de teste para uma classe que tivesse um atributo (fator) que poderia tomar 6 valores (6^1), 1 atributo que poderia levar 7 valores (7^1), outro atributo que poderia levar 2 valores (2^1), 1 atributo que poderia levar 9 valores (9^1). No sistema e no contexto do teste de aceitação, esta mesma amostra poderia ser usada para identificar cenários de teste (objetivos de teste) em uma abordagem de geração de casos de teste baseada em modelos (SANTIAGO JÚNIOR, 2011; SANTIAGO JÚNIOR; VIJAYKUMAR, 2012). Em ambos os casos, as séries de testes devem satisfazer o critério de teste *pairwise testing* ($t = 2$), onde todas as interações de fatores, 2 a 2, devem ser cobertas.

4.1.2 Hipóteses

A Tabela 4.2 ilustra as hipóteses, e os seus números de identificação, para a avaliação do custo relacionado ao tamanho dos conjuntos de teste, tempo de execução para a sua geração e análise de similaridade.

4.1.3 Variáveis

Com relação às variáveis envolvidas nesta experiência, destacam-se as variáveis independentes e variáveis dependentes. O primeiro tipo são aqueles que podem ser manipulados ou controlados durante o processo de julgamento e definição das causas das hipóteses (CAMPANHA et al., 2010). Para esta experiência, tais variáveis são o algoritmo/ferramenta para gerar *designs* combinatóriais, as instâncias/amostras uti-

Tabela 4.1 - Amostras para o experimento controlado: Instâncias.

i	Strength	Instance
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$
2	2	$2^3 3^1 4^1 2^2 3^1$
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$
9	3	$2^1 5^1 2^1 3^2 2^3$
10	3	$2^3 3^1 4^1 2^2 3^1$
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$
12	3	$3^2 2^1 4^1 2^4$
13	3	$2^2 3^2 4^2 5^2$
14	3	$2^2 3^2 4^2 5^2 6^2$
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$
17	4	$22^3 3^1 4^1 2^2 3^1$
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$
20	4	$2^2 3^2 4^2$
21	4	$2^2 3^2 4^2 5^2$
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$
23	5	$2^3 3^1 4^1 2^2 3^1$
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$
26	5	$2^2 3^2 4^2 5^2$
27	6	$2^2 3^2 4^2$

lizadas e a linguagem de programação em que os algoritmos foram implementados: Java (TTR, IPOG-F, IPO-TConfig), C++ (PICT) e C (*jenny*). Para as variáveis dependentes, pode-se considerar o resultado da manipulação das variáveis independentes (CAMPANHA et al., 2010). Para este estudo, identificaram-se o número de casos de teste gerados, o tempo de geração dos conjuntos de testes, e o resultado da análise de similaridade entre os conjuntos de testes.

4.1.4 Descrição do Experimento

Cada um dos algoritmos/ferramentas foi submetido a cada uma das 27 instâncias de teste (ver Tabela 4.1), um de cada vez. A saída de cada algoritmo/ferramenta, com o conjunto de testes gerada de acordo com cada instância, foi direcionada

Tabela 4.2 - Experimento 1 - Hipóteses

Hipóteses	IPOG-F	jenny	IPO-TConfig	PICT
Hipótese Nula - Não existe diferença de custo (tamanho do conjunto de teste) entre o TTR e outra solução	H1.0	H2.0	H3.0	H4.0
Hipótese Alternativa - Existe diferença de custo (tamanho do conjunto de teste) entre o TTR e outra solução	H1.0	H2.0	H3.0	H4.0
Hipótese Nula - Não existe diferença de custo (tempo para gerar o conjunto de teste) entre o TTR e outra solução	H5.0	H6.0	-	-
Hipótese Alternativa - Existe diferença de custo (tempo para gerar o conjunto de teste) entre o TTR e outra solução	H5.1	H6.1	-	-
Hipótese Nula - Não existe diferença de similaridade entre o conjunto de teste grada pelo TTR e outra solução	H7.0	H8.0	H9.0	H10.0
Hipótese Alternativa - Existe diferença de similaridade entre o conjunto de teste gerada pelo TTR e outra solução	H7.1	H8.1	H9.1	H10.1

para um arquivo de texto a ser gravado. Um ponto importante a ser ressaltado é que, além do próprio TTR, foi implementado para esse trabalho de mestrado, uma versão do IPOG-F. As outras ferramentas (PICT, *jenny* e IPO-TConfig) já foram implementadas e prontas para uso.

Para analisar o custo considerando o tamanho dos conjuntos de testes, verificou-se a quantidade de casos de teste gerados, ou seja, o número de linhas da matriz M, para cada instância/amostra.

Para a segunda perspectiva de custo (tempo para a geração dos conjuntos de testes), é necessário considerar que não foi possível medir o tempo de execução de alguns dos algoritmos/ferramentas analisados, uma vez que não é possível ter acesso ao seu código fonte. Contudo, foi possível medir o tempo de geração dos conjuntos de testes do TTR 1.1, da implementação feita nesse trabalho do IPOG-F, e do

algoritmo *jenny*. Para realizar esta medida de tempo, instrumentou-se cada uma das ferramentas e foi medido o tempo atual do computador antes e após a execução de cada algoritmo/ferramenta. Em todos os casos, foi usado um computador com processador Intel Core (i7-4790 CPU @ 3,60 GHz), 8 GB de RAM, executando o sistema operacional Microsoft Windows 7 Professional de 64 bits. O objetivo desta segunda análise é fornecer uma avaliação empírica do desempenho temporal dos algoritmos.

Para as duas medidas de custo, foi utilizada uma avaliação estatística apropriada, verificando a normalidade dos dados. A verificação da normalidade foi feita em três etapas: (i) usando o teste de *Shapiro-Wilk* (SHAPIRO; WILK, 1965) com um nível de significância $\alpha = 0{,}05$; (ii) verificando a inclinação da distribuição de frequências; e (iii) usando uma verificação gráfica por meio de gráficos Q-QPlot (KOHL, 2015) e histogramas. Assim, é possível obter uma maior confiança na conclusão sobre a normalidade dos dados em comparação com uma abordagem que se baseia apenas no teste de *Shapiro-Wilk*, considerando os efeitos de polarização[6] devidos ao comprimento das amostras. Como será discutido na Seção 4.1.6, em todos os casos e considerando todas as 6 primeiras hipóteses, os dados não eram normalmente distribuídos. Portanto, foi necessário o emprego do teste não paramétrico de *Wilcoxon* (*Signed Rank*) (KOHL, 2015) com nível de significância $\alpha = 0{,}05$. No entanto, se as amostras apresentam *ties*[7], é necessário a utilização de uma variação do teste de *Wilcoxon*, o *Wilcoxon Exact* (*Signed Rank*) (KOHL, 2015), adequado para tratar *ties* com nível de significância $\alpha = 0{,}05$.

Com relação à análise de similaridade, tomando como base o TTR 1.1, foi desenvolvido um programa em python onde busca-se as diferenças entre os conjuntos de testes geradas pelo TTR 1.1 e todos os outros algoritmos/ferramentas. O método de análise de similaridade é apresentado no Algoritmo 8.

Cada elemento $x \in X$ é a quantidade de casos de teste gerados via TTR 1.1 para cada instância i. Depois de obter o conjunto X, calcula-se L: um conjunto de valores ideiais onde cada elemento, $l \in L$, é 95% de um respectivo valor $x \in X$. O conjunto Y é referente aos casos de teste que são comuns a ambas implementações. Assim, cada elemento $y \in Y$ é a quantidade de casos de teste gerados pelo TTR 1.1 que são comuns as outras soluções, para cada instância i. Foi então calculada a distância

[6] Amostras Polarizadas são amostras que apresentam alguma "irregularidade" que tem como consequência um resultado tendencioso para o experimento.

[7] Ao comparar duas série de dados, em que uma determinada posição armazena exatamente o mesmo valor para ambas as séries, da-se o nome de *ties*.

Algorithm 8 Método construído para a análise de similaridade

1: obter o conjunto $X = \{x \quad | \quad x = |TS_i^{TTR}|\}$
2: calcular o conjunto $L = \{l \quad | \quad l = 0.95x, \quad x \in X\}$
3: obter o conjunto $Y = \{y \quad | \quad y = |TS_i^{TTR} \cap TS_i^{other}|\}$
4: $d_{LX} \leftarrow calcularDistanciaEuclidiana(n, L, X)$
5: $d_{YX} \leftarrow calcularDistanciaEuclidiana(n, Y, X)$
6: **if** $d_{YX} \leq d_{LX}$ **then**
7: Os conjuntos de teste X e Y são similares
8: **else**
9: Os conjuntos de teste X e Y não são similares

euclidiana em um espaço n-dimensional (n = 27 nesse estudo) considerando L e X, d_{LX} e considerando Y e X, d_{YX}. Concluí-se que os dois conjuntos de teste são similares se $d_{YX} \leq d_{LX}$. Consequentemente, o conjunto de valores ideais, L, ajuda a determinar a máxima distância euclidiana aceitável para considerar um conjunto Y (os casos de teste que o TTR 1.1 tem com outras soluções) similar ao conjunto X (casos de teste gerados pelo TTR 1.1).

4.1.5 Validade

Nesta seção, será discutido alguns aspectos relacionados à validade do experimento. Quanto à validade da conclusão, é importante considerar a confiabilidade das métricas. Neste estudo, as quantidades de casos de teste foram obtidas de forma automática pelos algoritmos/ferramentas, e é possível obter os mesmos resultados, no caso de replicação deste experimento por outros pesquisadores. Para minimizar o impacto da ordem de entrada de fatores e níveis e obter resultados mais consistentes para a análise estatística, foram gerados casos de teste com 3 variações na ordem de entrada de fatores e níveis e levou-se em consideração a média destes 3 valores para os testes estatísticos.

Espara-se que a validade da conlusão com relação ao tempo de geração dos casos de testes seja a mesma em caso de replicação desse experimento. Mais uma vez, cada ferramenta foi executada 3 vezes e os valores médios para os testes foram considerados. No entanto, espera-se que uma replicação deste estudo forneça diferentes resultados de tempo simplesmente porque esses resultados dependem da configuração do computador (processador, memória, sistema operacional) usada para executar os algoritmos/ferramentas. Mas, para esta perspectiva de tempo, o desempenho da melhor solução (*jenny*) foi muito melhor do que a segunda melhor abordagem (IPOG-F), e consideravelmente melhor do que o algoritmo proposto nesse trabalho

(TTR 1.1), que era o mais fraco. Assim, não pode-se esperar uma validade diferente da conclusão

Espera-se a mesma validade de conclusão, em caso de replicação, para a análise de similaridade. A linha de fundo é que a análise de similaridade é baseada na quantidade de casos de teste gerados e, portanto, o mesmo raciocínio apresentado anteriormente relacionado com o custo na perspectiva do tamanho das suítes de teste se aplica aqui. Os casos de testes também foram gerados 3 vezes para cada instância e para todos os algoritmos/ferramentas, mas como o intuíto é resgatar os casos de teste comuns (TTR 1.1 e outro algoritmo/ferramenta) foi necessário realizar uma análise separada, por execução, ao invés de considerar um valor médio dos resultados. Para todas as 3 avaliações (custo/tamanho, custo/tempo, similaridade), 10 hipóteses foram estatisticamente avaliadas por meio de testes de hipóteses e verificação de similaridade via distância euclidiana.

As ameaças à validade interna comprometem a confiança ao afirmar que há uma relação entre variáveis dependentes e independentes. Não houve fatores que interferissem nessa relação porque os participantes, ou seja, as amostras/instâncias, foram selecionados aleatoriamente, não houve eventos imprevistos para interromper a coleta de métricas uma vez iniciada e a geração de casos de teste seguiu rigorosamente os algoritmos implementados nas ferramentas. Da mesma forma, a validade da construção também foi assegurada, uma vez que os algoritmos/ferramentas tradicionais de *designs* combinatóriais foram usados para serem comparados ao TTR 1.1.

Finalmente, as ameaças à validade externa comprometem a confiança em afirmar que os resultados do estudo podem ser generalizados para e entre indivíduos, contextos e sob a perspectiva temporal. Basicamente, pode-se dividir as ameaças à validade externa em duas categorias: ameaças à população e ameaças ecológicas.

As ameaças à população referem-se à significância do conjunto de amostras da população utilizada no estudo. Para esse estudo, os *strengths* e a variedade de fatores e níveis que compõem as instâncias são os pontos determinantes para a caracterização dessa ameaça. É possível observar que para tal estudo, a possibilidade de combinação de *strengths* e fatores/níveis é literalmente infinita. No entanto, ao contrário de outros estudos informais (CZERWONKA, 2006) em que se concentra mais no *pairwise testing* (*strength* = 2), 70% dos valores de *strength* que foram usados nesse estudo são maiores do que 2, além disso, há casos com até 20 fatores, um fator que pode levar até 11 (Ver exemplo 8 na Tabela 4.1). Por isso, pode-se considerar que a escolha do conjunto de amostras é mais significativa do que em outros estudos na literatura.

As ameaças ecológicas referem-se ao grau em que os resultados podem ser generalizados entre diferentes configurações. Os efeitos de interação (por exemplo, realizar um pré-teste com os participantes da experiência) e o efeito *Hawthorne* (devido aos participantes simplesmente sentirem-se estimulados por saber que participam de uma experiência inovadora) são alguns dos tipos desta ameaça. Os participantes dessa experiência são os exemplos de teste e, portanto, este tipo de ameaça não se aplica a esse experimento.

4.1.6 Resultados

4.1.6.1 Custos: Tamanho dos Conjuntos de Testes e Tempo de Geração

Como já foi dito, a experiência relatada neste trabalho teve como objetivo avaliar o aspecto relacionado ao custo, levando em conta os algoritmos/ferramentas TTR 1.1, PICT, *jenny*, IPOG-F e IPO-TConfig: quantidade de casos de teste gerados e tempo para a geração dos conjuntos de testes. Além disso, uma análise de similaridade foi realizada para verificar se os conjuntos de casos de teste criados via TTR eram semelhantes aos de outros algoritmos/ferramentas.

As hipóteses (Seção 4.1.2) 1 a 4 referem-se à primeira avaliação de custo, a quantidade de casos de teste, enquanto as hipóteses de 5 a 6 referem-se a segunda perspectiva de custo, tempo para a geração dos conjuntos de testes. É importante enfatizar que esse tempo está relacionado com a geração dos conjuntos de teste. As hipóteses de 7 a 10 estão relacionadas com a análise de similaridade entre os conjuntos de casos de teste.

Considerando o custo relacionado à quantidade de casos de teste, conforme descrito na Seção 4.1, a normalidade dos dados foi determinada através de 4 métodos: via teste *Shapiro-Wilk* com nível de significância $\alpha = 0.05$, O valor *skewness*, e verificação gráfica por meio de gráficos Q-QPlot e histogramas. Nenhum dos dados relacionados às quatro hipóteses foram normalmente distribuídos: na Tabela 4.3 podemos verificar que os valores de *skewness* das amostras estão distantes de zero e os *p-values* do teste de *Shapiro-Wilk* são inferiores a $\alpha = 0.05$. A Figura 4.1 apresenta os gráficos Q-QPlots de todas as amostras, e nenhuma delas apresenta distribuição normal uma vez que em nenhum dos casos, a linha pontilhada descreveu curva similar a linha reta traçada no gráfico. A Figura 4.2 apresenta os gráficos histogramas das amostras, e também, em nenhum caso há distribuíção normal dos dados considerando que em nenhum deles apresenta as barras dispostas de modo a formar uma espécie de Gaussiana. Portanto, é fato que nenhuma das amostras apresenta distribuíção

normal.

Tabela 4.3 - Experimento 1 - Valores relacionados ao teste de normalidade dos dados da quantidade de casos de testes: *skewness* e *Shapiro-Wilk*.

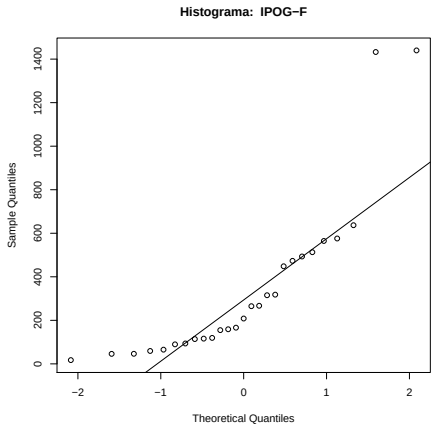
Amostras	*skewness*	*Shapiro-Wilk*
IPOG-F	$2,074309$	$1,566e-05$
jenny	$2,145283$	$7,965e-06$
IPO-TConfig	$2,129089$	$7,794e-06$
PICT	$2,235561$	$5,352e-06$
TTR	$2,305014$	$4,231e-06$

Portanto, foi aplicado o teste não paramétrico *Wilcoxon Asymptotic* (*Signed Rank*) (KOHL, 2015) com nível de significância $\alpha = 0.05$. A Tabela 4.4 mostra os resultados e o valor médio total ($\bar{x}$) em relação ao número de casos de teste gerados pelas cinco soluções. Devido ao fato de todos os algoritmos serem sensíveis à ordem de entrada de fatores e níveis, cada instância foi considerada três vezes: uma na ordem mostrada na Tabela 4.4, outra com o primeiro e o último fatores trocados e outra com o segundo e penúltimo fatores trocados. Assim, os valores que aparecem devido a cada solução é uma média dessas três submissões. Por exemplo, na instância/amostra 1, a média das três execuções do *jenny* foi de 71.67 enquanto que a média de TTR foi de 63. A tabela 4.5 mostra os *p-values* relacionados com esta avaliação.

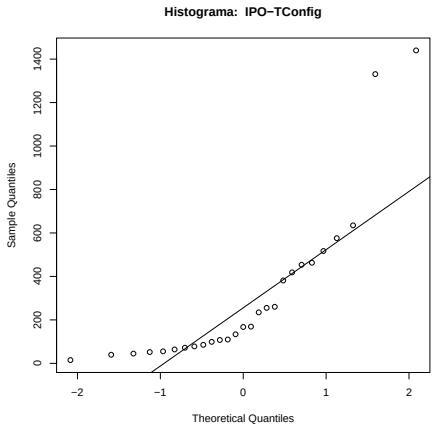
De acordo com esses resultados, observou-se que todas as quatro hipóteses nulas (H1.0 a H4.0) foram rejeitadas porque os *p-values* estão abaixo de 0.05. Portanto, há diferença no custo, considerando a quantidade de casos de teste gerados através dos algoritmos/ferramentas. Como o valor médio total ($\bar{x}$) do TTR é menor do que qualquer outro valor médio atribuído a outras soluções (Veja a Figura 4.3, a barra referente a mediana do TTR 1.1 é menor que todos os outros algoritmos/ferramentas), concluímos que TTR é a melhor alternativa para gerar um conjunto de testes de custo mais baixo, em termos de tamanho de *suite* de teste gerada.

Com relação ao custo sob a perspectiva do tempo para gerar os conjuntos de teste, seguiu-se o mesmo procedimento já descrito para verificar a normalidade dos dados. Como no caso anterior, os dados relacionados às hipóteses 5 a 6 também não apresentaram distribuição normal: na Tabela 4.6 podemos verificar que os valores de *skewness* das amostras estão distantes de zero e os *p-values* do teste de *Shapiro-Wilk* são inferiores a $\alpha = 0{,}05$. A Figura 4.4 apresenta os Q-QPlots de todas as amostras, e nenhuma delas apresenta distribuição normal. A Figura 4.5 representa os histogra-

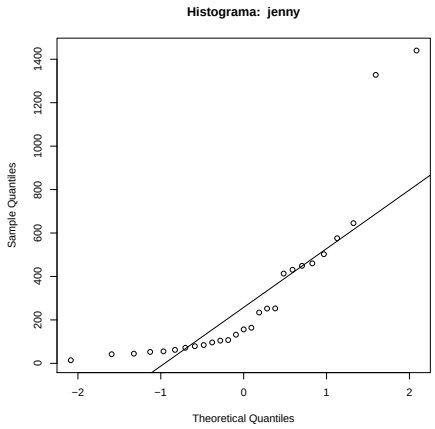
Figura 4.1 - Experimento 1 - Gráficos Q-QPLot dos dados: quantidade de casos de teste.



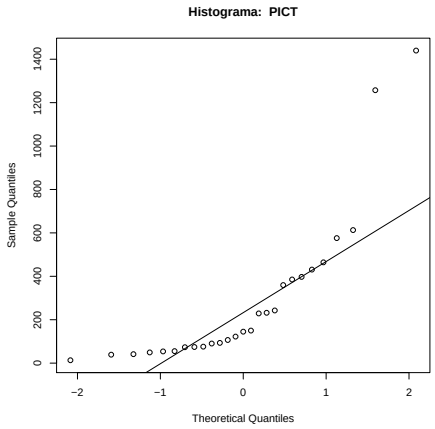
(a)



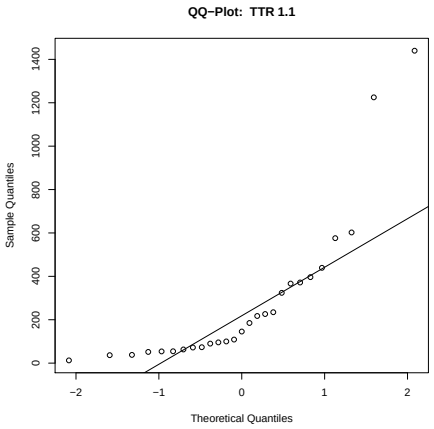
(b)



(c)

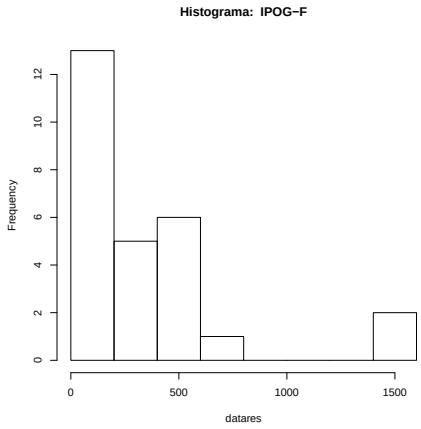


(d)

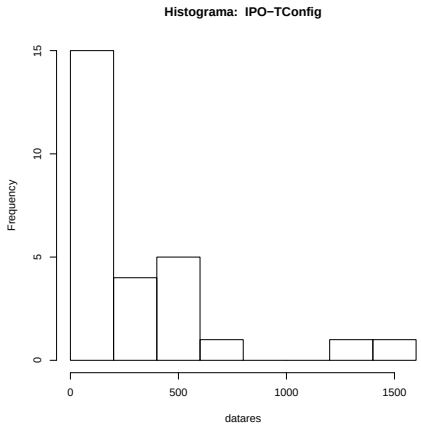


(e)

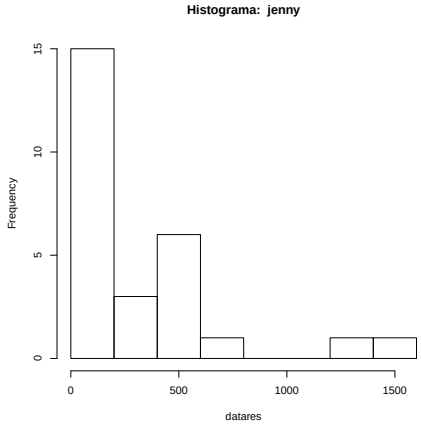
Figura 4.2 - Experimento 1 - Histogramas dos dados: tempo de geração dos casos de testes.



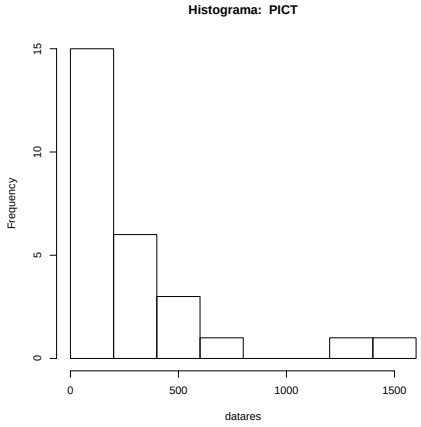
(a)



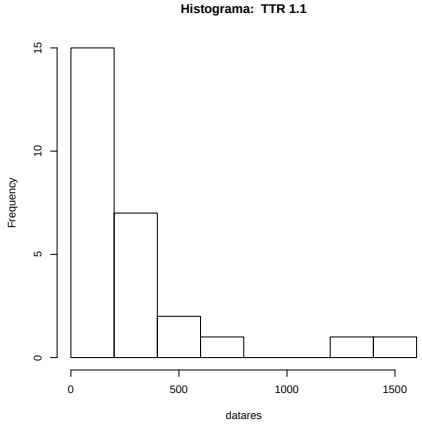
(b)



(c)

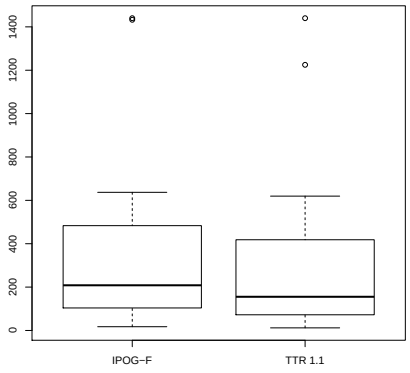


(d)

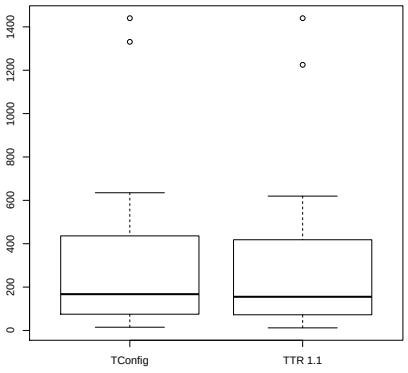


(e)

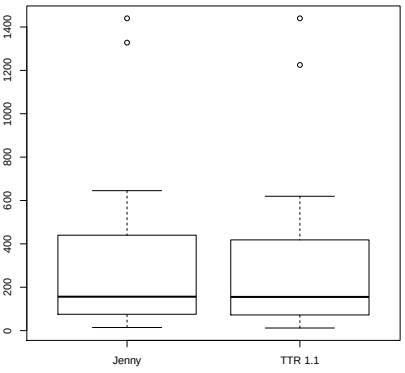
Figura 4.3 - Gráficos BoxPlot dos dados em relação ao tamanho dos conjuntos de testes gerados para o TTR 1.1



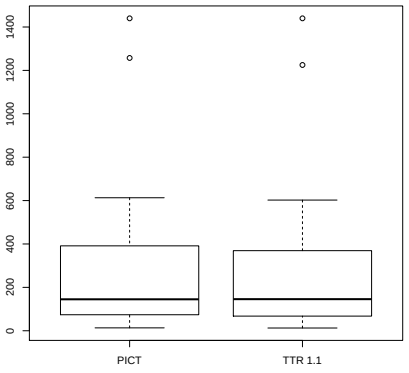
(a)



(b)



(c)



(d)

Tabela 4.4 - Experimento 1 - Custo relacionado ao tamanho dos conjuntos de teste: resultados e valores médios.

i	Strength	Instância	*jenny*	IPO-TConfig	PICT	IPOG-F	TTR
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$	71,67	72,00	76,00	89,67	63
2	2	$2^3 3^1 4^1 2^2 3^1$	14,33	15,00	13,33	17,33	12,67
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$	79,00	78,00	74,33	93,67	72,00
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$	55,33	55,33	54,00	65,33	54,00
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$	62,00	64,00	55,33	119,33	54,67
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$	84,33	85,33	73,33	159,00	73,33
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$	104,67	109,67	93,33	208,33	95,67
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$	156,33	167,33	145,00	318,00	185,00
9	3	$2^1 5^1 2^1 3^2 2^3$	52,67	52,00	49,33	59,33	51,67
10	3	$2^3 3^1 4^1 2^2 3^1$	44,67	45,00	39,00	46,33	38,00
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$	96,33	99,00	90,33	116,00	90,00
12	3	$3^2 2^1 4^1 2^4$	42,00	39,67	41,33	46,00	36,67
13	3	$2^2 3^2 4^2 5^2$	132,00	133,67	122,33	155,00	108,67
14	3	$2^2 3^2 4^2 5^2 6^2$	252,33	260,33	229,00	315,33	217,33
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$	431,00	381,33	385,33	473,33	371,67
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$	253,00	255,33	242,67	265,00	226,33
17	4	$2^3 3^1 4^1 2^2 3^1$	107,33	107,67	106,67	114,00	100,00
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$	460,33	463,00	431,00	513,00	324,00
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$	449,00	453,67	359,67	493,33	396,33
20	4	$2^2 3^2 4^2$	164,00	169,00	150,00	166,33	145,67
21	4	$2^2 3^2 4^2 5^2$	502,67	517,00	464,33	564,67	439,33
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$	413,33	418,67	397,33	448,33	366,67
23	5	$2^3 3^1 4^1 2^2 3^1$	234,00	234,67	232,00	267,00	234,67
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$	1328,00	1331,00	1257,33	1432,67	1225,00
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$	645,33	635,00	613,00	637,00	602,33
26	5	$2^2 3^2 4^2 5^2$	1440,00	1440,00	1440,00	1440,00	1440,00
27	6	$2^2 3^2 4^2$	576,00	576,00	576,00	576,00	576,00
$\bar{x}$			305,62	305,88	289,31	340,72	281,51

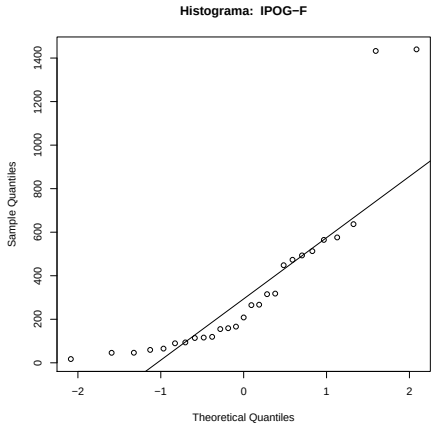
Tabela 4.5 - Experimento 1 - Custo relacionado ao tamanho dos conjuntos de teste: *Asymptotic Wilcoxon*

Hipóteses	*p-value*
1: TTR $\leftrightarrow$ IPOG-F	$5,96e-08$
2: TTR $\leftrightarrow$ *jenny*	$1,508e-05$
3: TTR $\leftrightarrow$ IPO-TConfig	$1,264e-05$
4: TTR $\leftrightarrow$ PICT	$0,02294$

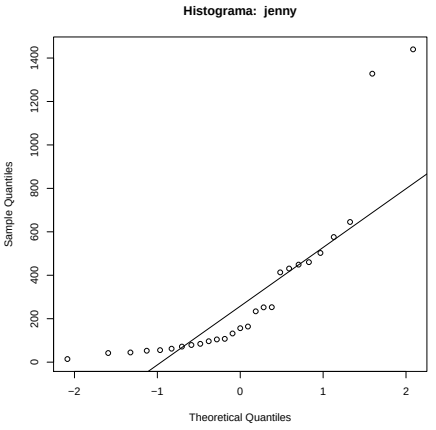
mas das amostras, e também, em nenhum caso há distribuição normal dos dados. Portanto, é fato que nenhuma das amostras apresenta distribuição normal.

Assim, mais uma vez utilizou-se o teste não paramétrico *Wilcoxon assintótico* (*Signed Rank*) (KOHL, 2015) com nível de significância $\alpha = 0{,}05$. A Tabela 4.7 mostra os resultados e o valor médio total ($\bar{x}$) relacionado com o tempo (s) para gerar os

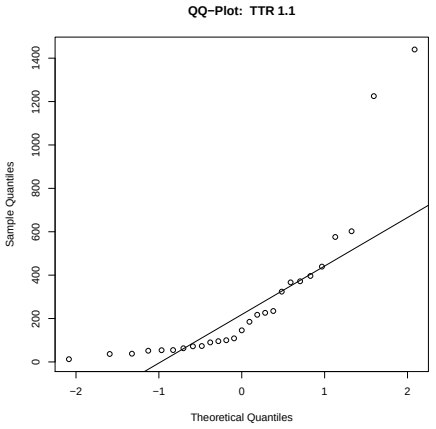
Figura 4.4 - Experimento 1 - Gráficos Q-QPLot dos dados do IPOG-F: tempo de geração dos casos de testes.



(a)

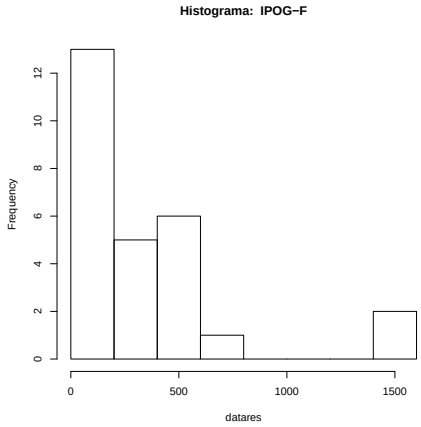


(b)

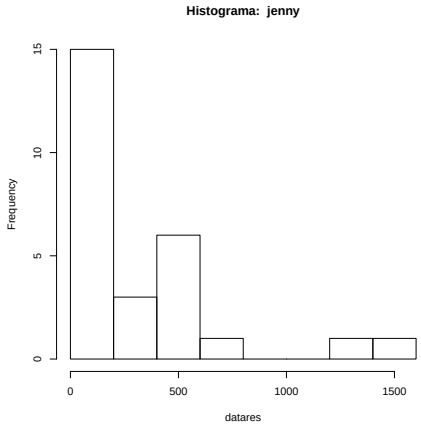


(c)

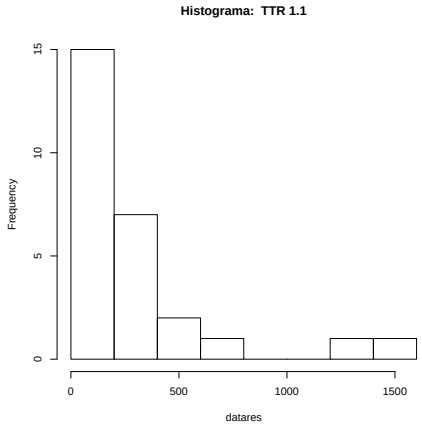
Figura 4.5 - Experimento 1 - Histograma dos dados do IPOG-F: tempo de geração dos casos de testes.



(a)



(b)



(c)

Tabela 4.6 - Experimento 1 - Valores relacionados ao teste de normalidade dos dados de tempo de geração dos casos de testes: *skewness* e *Shapiro-Wilk*.

Amostras	*skewness*	*Shapiro-Wilk*
IPOG-F	$4,6995$	$4,241e-10$
jenny	$2,207532$	$2,473e-05$
TTR	$5,259257$	$3,985e-11$

conjuntos de testes. Por causa da questão da sensibilidade à ordem de entrada de fatores e níveis, foram geradas três vezes os casos de testes, e o valor que aparece em cada linha da Tabela 4.7 é uma média dessas três submissões. A Tabela 4.8 mostra os *p-values* relacionados com esta avaliação.

Estes resultados mostram novamente que as duas hipóteses nulas (H5.0 e H6.0) foram rejeitadas porque os *p-values* são inferiores a 0.05. Portanto, há diferença no custo, considerando o tempo para gerar o conjunto de teste através dos algoritmos/ferramentas. O valor médio do TTR 1.1 é o maior de todos os valores (Veja a Figura 4.6, em que a barra que representa a mediana do TTR 1.1 é maior que a barra correspondente as demais soluções avaliadas) e, ao contrário da primeira avaliação de custo, o TTR 1.1 apresentou o desempenho mais fraco nesta segunda avaliação. Um fator que possa ter contribuído com o pior desempenho do TTR 1.1 quanto a esse aspecto é a linguagem de programação utilizada: o TTR 1.1 foi implementado em Java e o *jenny* foi desenvolvido em C. Inclusive, a maneira com que o algoritmo foi implementado também influência fortemente no desmepenho do algoritmo. É um fato que este último é uma linguagem muito mais apropriada em relação aos aspectos em tempo real. Porém, o IPOG-F também foi implementado em Java. A explicação para IPOG-F superar o TTR 1.1 é que IPOG-F faz comparações a fim de achar a melhor solução local (é um algoritmo guloso) apenas em determinados casos de testes. Depois disso, o que ele faz é completar as *t-tuplas* restantes, sem se preocupar com a quantidade de *t-tuplas* que foram cobertas em cada iteração. Isso faz com que o algoritmo IPOG-F seja mais rápido a partir de um determinado ponto, no entanto, ele produz uma maior quantidade de casos de teste, em comparação com TTR, devido a este recurso. Além do mais, é importante ressaltar que o tempo de execução dos conjuntos de teste é mais relevante em relação ao processo de Teste de Software como um todo, comparado ao tempo de geração dos conjuntos de testes.

Tabela 4.7 - Experimento 1 - Custo relacionado ao Tempo (s) para gerar os conjuntos de testes: resultados e valores médios

i	Strength	Instance	*jenny*	IPOG-F	TTR
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$	0,074	0,163	0,115
2	2	$2^3 3^1 4^1 2^2 3^1$	0,026	0,067	0,080
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$	0,090	0,150	0,173
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$	0,075	0,077	0,130
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$	0,099	0,163	0,991
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$	0,136	0,302	4,400
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$	0,203	0,595	18,999
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$	0,313	2,430	1240,164
9	3	$2^1 5^1 2^1 3^2 2^3$	0,075	0,131	0,947
10	3	$2^3 3^1 4^1 2^2 3^1$	0,055	0,157	0,732
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$	0,098	0,275	1,783
12	3	$3^2 2^1 4^1 2^4$	0,079	0,121	0,731
13	3	$2^2 3^2 4^2 5^2$	0,126	0,355	4,601
14	3	$2^2 3^2 4^2 5^2 6^2$	0,347	2,550	172,537
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$	0,944	18,313	34664,835
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$	0,250	1,507	56,274
17	4	$2^3 3^1 4^1 2^2 3^1$	0,118	0,484	12,128
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$	0,498	3,301	143,801
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$	0,476	4,843	145,860
20	4	$2^2 3^2 4^2$	0,114	0,155	0,463
21	4	$2^2 3^2 4^2 5^2$	0,554	3,824	207,121
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$	0,438	3,073	116,216
23	5	$2^3 3^1 4^1 2^2 3^1$	0,232	1,394	42,661
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$	1,601	87,788	1271,550
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$	0,709	18,094	316,005
26	5	$2^2 3^2 4^2 5^2$	0,723	0,424	0,193
27	6	$2^2 3^2 4^2$	0,376	0,106	0,129
$\bar{x}$			0,327	5,587	1423,097

Tabela 4.8 - Experimento 1 - Custo relacionado ao tempo (s) para gerar os conjuntos de teste: *Asymptotic Wilcoxon*

Hipóteses	*p-value*
5: TTR $\leftrightarrow$ IPOG-F	$3,204e-07$
6: TTR $\leftrightarrow$ *jenny*	$5,215e-07$

4.1.6.2 Análise de Similaridade

Seguindo o método descrito na Seção 4.1, foi realizada uma análise para verificar se os conjuntos de casos de teste gerados pela versão 1.2 do TTR foram semelhantes

aos conjuntos de teste gerados pelo IPOG-F, IPO-TConfig, *jenny*, e PICT. Também foram feitas 3 execuções de cada algoritmo/ferramenta, mas como a idéia principal é calcular a distância euclidiana comparando o número de casos de teste gerados pelo TTR (set X) com o número de casos de teste gerados por outras soluções que são comuns ao TTR (set Y), onde foi considerado cada execução separadamente.

Na Tabela 4.9 é possível observar apenas os resultados da primeira execução. A coluna TTR é o conjunto X e as outras colunas são conjuntos Y. Assim, para a primeira instância, *jenny*, IPO-TConfig e IPOG-F não têm nenhum caso de teste em comum com o TTR, enquanto 11 casos de teste gerados pelo PICT também foram gerados pelo TTR. Foi então calculado o conjunto de valores ideais, L, e as distâncias euclidianas (d_{LX}, d_{YX}), como apresentado no Algoritmo 8.

Com base na tabela 4.10, é claro que todas as hipóteses nulas de 7 a 10 (H7.0 H10.0) são rejeitadas. Em outras palavras, os conjuntos de teste gerados via TTR não são semelhantes a nenhum dos conjuntos de casos de teste gerados pelas outras abordagens. O que é positivo, já que pode ser uma indicação de que o TTR tende a estimular diferentes partes do SUT (*Software Under Test*) em relação aos outros algoritmos/ferramentas.

4.2 Experimento Controlado 2: Custo e Similaridade

Conforme observações na Seção 4.1, o tempo que é mais relevante, considerando o processo de teste como um todo, é o tempo para executar os conjuntos de teste, e não o tempo para gerar os conjuntos de casos de teste. Por isso, nesse segundo experimento controlado, comparou-se o TTR, versão 1.2, com os mesmos quatro algoritmos/ferramentas anteriores, mas considerando somente a perspectiva de custo relacionada ao tamanho dos conjuntos de teste. A análise de similaridade também foi realizada. Além disso, foi considerada a versão final, versão 1.2, do algoritmo TTR.

Esse experimento controlado foi conduzido de acordo com o experimento descrito na Seção 4.1. Dessa maneira, o experimento controlado é o mesmo, onde a única diferença é o fato de não ser considerado o *tempo de execução* do algoritmo. Naturalmente, o tempo para gerar cada conjunto de teste não é uma variável dependente nesse segundo experimento e, dessa forma, não é necessário considerar observações relacionadas a essa questão na Seção de Validação. As instâncias consideradas foram as mesmas 27 instâncias ilustradas na Tabela 4.1, e, igualmente, cada instância foi executada de três maneiras diferentes por cada um dos algoritmos/ferramentas

Tabela 4.9 - Experimento 1 - Análise de similaridade: conjuntos X e Y da primeira execução.

i	Strength	Instância	*jenny*	IPO-TConfig	IPOG-F	PICT	TTR
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$	0	0	0	11	63
2	2	$2^3 3^1 4^1 2^2 3^1$	0	0	0	4	12
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$	0	0	2	10	72
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$	1	1	1	15	54
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$	2	4	3	49	56
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$	8	3	4	57	75
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$	1	2	2	52	100
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$	26	2	2	106	185
9	3	$2^1 5^1 2^1 3^2 2^3$	15	13	14	29	53
10	3	$2^3 3^1 4^1 2^2 3^1$	12	7	14	15	36
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$	11	5	5	35	90
12	3	$3^2 2^1 4^1 2^4$	8	3	7	17	39
13	3	$2^2 3^2 4^2 5^2$	7	11	8	25	109
14	3	$2^2 3^2 4^2 5^2 6^2$	9	4	5	39	217
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$	2	2	3	65	372
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$	62	42	48	89	226
17	4	$2^3 3^1 4^1 2^2 3^1$	23	18	22	40	99
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$	48	29	49	146	413
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$	73	58	59	146	397
20	4	$2^2 3^2 4^2$	44	39	68	80	146
21	4	$2^2 3^2 4^2 5^2$	58	46	63	168	445
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$	104	90	100	180	366
23	5	$2^3 3^1 4^1 2^2 3^1$	66	75	67	99	233
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$	102	0	254	446	1228
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$	129	148	170	255	601
26	5	$2^2 3^2 4^2 5^2$	1440	1440	1440	1440	1440
27	6	$2^2 3^2 4^2$	576	576	576	576	576

Tabela 4.10 - Experimento 1 - Análise de similaridade: distâncias euclidianas. Legenda: max = máxima distância euclidiana permitida.

	max	IPOG-F	*jenny*	IPO-TConfig	PICT
Execução 1	115,63	1598,15	1507,63	1084,56	1390,59
Execução 2	113,70	1411,37	1457,43	975,65	1332,85
Execução 3	115,57	1499,67	1441,89	1096,42	1444,73

avaliados. Dessa forma, a validade e a descrição do experimento são as mesmas, e podem ser consultadas respectivamente nas Seções 4.1.4 e 4.1.5.

4.2.1 Hipóteses

A Tabela 4.11 ilustra as hipóteses, e os seus números de identificação, para a avaliação do custo relacionado ao tamanho dos conjuntos de teste, tempo de execução para a sua geração e análise de similaridade.

Tabela 4.11 - Experimento 2 - Hipóteses

Hipóteses	IPOG-F	jenny	IPO-TConfig	PICT
Hipótese Nula - Não existe diferença de custo (tamanho do conjunto de teste) entre o TTR e outra solução	H11.0	H12.0	H13.0	H14.0
Hipótese Alternativa - Existe diferença de custo (tamanho do conjunto de teste) entre o TTR e outra solução	H11.1	H12.1	H13.1	H14.1
Hipótese Nula - Não existe diferença de similaridade entre o conjunto de teste grada pelo TTR e outra solução	H15.0	H16.0	H17.0	H18.0
Hipótese Alternativa - Existe diferença de similaridade entre o conjunto de teste gerada pelo TTR e outra solução	H15.1	H16.1	H17.1	H18.1

4.2.2 Resultados

4.2.2.1 Custo: Tamanho dos Conjuntos de Testes

As hipóteses (Seção 4.2.1) de 11 a 14 referem-se à primeira avaliação de custo: quantidade de casos de teste, enquanto as hipóteses de 15 a 18 estão relacionadas com a análise de similaridade entre os conjuntos de testes.

Considerando o custo relacionado à quantidade de casos de teste, conforme descrito na Seção 4.1, nenhum dos dados relacionados às quatro hipóteses foram normalmente distribuídos: na Tabela 4.12 é possível verificar que os valores de *skewness* das amostras estão distantes de zero e os *p-values* do teste de *Shapiro-Wilk* são inferiores a $\alpha = 0.05$. A Figura 4.7 representa o Q-QPlot da amostra referente ao TTR 1.2 (as demais amostras estão representadas na Figura 4.1), e ela não apresenta distribuição normal. A Figura 4.8 representa o histograma da amostra referente ao TTR 1.2 (as demais amostras estão representadas na Figura 4.2), e também, em nenhum caso há distribuíção normal dos dados. Portanto, é fato que nenhuma das amostras apresenta distribuíção normal.

Aplicou-se então o teste não paramétrico *Wilcoxon Asymptotic* (*Signed Rank*) (KOHL, 2015) com nível de significância $\alpha = 0.05$. A Tabela 4.13 mostra os resultados e o valor médio total ($\bar{x}$) em relação ao número de casos de teste gerados

Tabela 4.12 - Experimento 2 - Valores relacionados ao teste de normalidade dos dados da quantidade de casos de testes: *skewness* e *Shapiro-Wilk*

Amostras	*skewness*	*Shapiro-Wilk*
IPOG-F	$2,074309$	$1,566e-05$
jenny	$2,129089$	$7,965e-06$
IPO-TConfig	$2,145283$	$7,794e-06$
PICT	$2,235561$	$5,352e-06$
TTR	$2,210743$	$6,357e-06$

pelas cinco soluções. Devido ao fato de que todos os algoritmos serem sensíveis à ordem de entrada de fatores e níveis, cada instância foi considerada três vezes: uma na ordem mostrada na Tabela 4.13, outra com o primeiro e o último fatores trocados e outra com o segundo e penúltimo fatores trocados. Assim, os valores que aparecem devido a cada solução é uma média dessas três submissões. Por exemplo, na instância/amostra 1, a média das três execuções do *jenny* foi de 71,67 enquanto que a média de TTR foi de 63. A tabela 4.14 mostra os *p-values* relacionados com esta avaliação.

De acordo com esses resultados, é fato que todas as quatro hipóteses nulas (H10,0 a H14.0) foram rejeitadas porque os *p-values* estão abaixo de 0.05. Portanto, há diferença no custo, considerando a quantidade de casos de teste gerados através dos algoritmos/ferramentas. Como o valor médio total ($\bar{x}$) do TTR é menor do que qualquer outro valor médio atribuído a outras soluções (Veja a Figura 4.9, a barra referente a média do TTR é menor que todos os outros algoritmos/ferramentas), concluímos que TTR é a melhor alternativa para gerar um conjunto de testes de custo mais baixo, em termos de tamanho de *suite* de teste gerada.

4.2.2.2 Análise de Similaridade

Seguindo o método descrito na Seção 4.1, foi realizada uma análise para verificar se os conjuntos de casos de teste gerados pela versão 1.2 do TTR são semelhantes aos conjuntos de teste gerados pelo IPOG-F, IPO-TConfig, jenny, E PICT. Aqui também foi realizada 3 execuções de cada algoritmo/ferramenta, mas como a idéia principal é calcular a distância euclidiana comparando o número de casos de teste gerados pelo TTR (set X) com o número de casos de teste gerados por outras soluções que são comuns ao TTR (set Y), foi considerado cada execução separadamente.

Na Tabela 4.15 é possível observar apenas os resultados da primeira execução. A coluna TTR é o conjunto X e as outras colunas são conjuntos Y. Assim, para a

Tabela 4.13 - Experimento 2 - Custo relacionado ao tamanho dos conjuntos de teste: resultados e valores médios TTR 1.2.

i	Strength	Instâncias	jenny	IPO-TCONFIG	PICT	IPOG	TTR
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$	71,67	72,00	76,00	89,67	65
2	2	$2^3 3^1 4^1 2^2 3^1$	14,33	15,00	13,33	17,33	12,00
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$	79,00	78,00	74,33	93,67	72,33
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$	55,33	55,33	54,00	65,33	54,00
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$	62,00	64,00	55,33	119,33	55,33
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$	84,33	85,33	73,33	159,00	72,67
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$	104,67	109,67	93,33	208,33	99,00
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$	156,33	167,33	145,00	318,00	155,33
9	3	$2^1 5^1 2^1 3^2 2^3$	52,67	52,00	49,33	59,33	50,00
10	3	$2^3 3^1 4^1 2^2 3^1$	44,67	45,00	39,00	46,33	39,67
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$	96,33	99,00	90,33	116,00	91,00
12	3	$3^2 2^1 4^1 2^4$	42,00	39,67	41,33	46,00	40,00
13	3	$2^2 3^2 4^2 5^2$	132,00	133,67	122,33	155,00	111,33
14	3	$2^2 3^2 4^2 5^2 6^2$	252,33	260,33	229,00	315,33	228,00
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$	431,00	381,33	385,33	473,33	372,00
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$	253,00	255,33	242,67	265,00	230,00
17	4	$2^3 3^1 4^1 2^2 3^1$	107,33	107,67	106,67	114,00	103,00
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$	460,33	463,00	431,00	513,00	418,33
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$	449,00	453,67	422,67	493,33	426,67
20	4	$2^2 3^2 4^2$	164,00	169,00	150,00	166,33	147,00
21	4	$2^2 3^2 4^2 5^2$	502,67	517,00	464,33	564,67	455,00
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$	413,33	418,67	397,33	448,33	388,67
23	5	$2^3 3^1 4^1 2^2 3^1$	234,00	234,67	232,00	267,00	232,67
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$	1328,00	1331,00	1257,33	1432,67	1225,00
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$	645,33	635,00	613,00	637,00	616,67
26	5	$2^2 3^2 4^2 5^2$	1440,00	1440,00	1440,00	1440,00	1440,00
27	6	$2^2 3^2 4^2$	576,00	576,00	576,00	576,00	576,00
$\bar{x}$			305,62	305,88	291,64	340,72	288,02

Tabela 4.14 - Experimento 2 - Custo relacionado ao tamanho dos conjuntos de teste: *Asymptotic Wilcoxon* TTR 1.2

Hipóteses	*p-value*
1: TTR $\leftrightarrow$ IPOG-F	$5,96e-08$
2: TTR $\leftrightarrow$ *jenny*	$5,96e-08$
3: TTR $\leftrightarrow$ IPO-TConfig	$1,192e-07$
4: TTR $\leftrightarrow$ PICT	$0,03067$

primeira instância, *jenny*, IPO-TConfig e IPOG-F não têm nenhum caso de teste em comum com o TTR, enquanto 11 casos de teste gerados pelo PICT também foram gerados pelo TTR. Foi calculado o conjunto de valores ideais, L, e as distâncias euclidianas (d_{LX}, d_{YX}), como apresentado no Algoritmo 8.

Com base na tabela 4.16, é claro que todas as hipóteses nulas de 15 a 18 (H15.0

H18.0) são rejeitadas. Em outras palavras, os conjuntos de teste gerados via TTR não são semelhantes a nenhum dos conjuntos de casos de teste gerados pelas outras abordagens. Isto pode ser uma indicação de que o TTR tende a estimular diferentes partes do SUT (*Software Under Test*) em relação aos outros algoritmos/ferramentas.

Tabela 4.15 - Experimento 2 - Análise de similaridade: conjuntos X e Y da primeira execução.

i	Strength	Instância	*jenny*	IPO-TConfig	IPOG-F	PICT	TTR
1	2	$6^1 7^1 2^1 4^1 3^1 7^1 4^1 9^1$	1	1	1	14	65
2	2	$2^3 3^1 4^1 2^2 3^1$	1	0	0	4	12
3	2	$9^1 8^1 2^1 4^1 5^1 6^1 7^1 2^1$	0	0	0	7	72
4	2	$3^2 2^1 4^1 5^1 6^1 9^1 2^1$	1	1	1	16	54
5	2	$2^2 3^2 4^2 5^2 6^2 7^2$	2	1	2	49	55
6	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2$	6	2	1	45	72
7	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2$	2	1	0	42	99
8	2	$2^2 3^2 4^2 5^2 6^2 7^2 8^2 9^2 10^2 11^2$	30	2	2	94	146
9	3	$2^1 5^1 2^1 3^2 2^3$	10	8	1	24	50
10	3	$2^3 3^1 4^1 2^2 3^1$	7	5	1	19	39
11	3	$2^1 5^1 6^1 2^2 3^1 2^2$	5	3	8	28	91
12	3	$3^2 2^1 4^1 2^4$	4	8	8	13	41
13	3	$2^2 3^2 4^2 5^2$	10	8	4	27	108
14	3	$2^2 3^2 4^2 5^2 6^2$	10	10	9	58	229
15	3	$2^2 3^2 4^2 5^2 6^2 7^2$	15	10	7	204	372
16	4	$5^1 3^1 2^2 3^1 2^1 4^1 3^1$	50	31	16	83	229
17	4	$2^3 3^1 4^1 2^2 3^1$	23	19	48	36	104
18	4	$2^1 5^1 2^1 4^1 5^1 3^1 4^1 2^1$	68	43	19	151	420
19	4	$3^2 2^1 4^1 5^1 6^1 2^2$	105	81	58	188	432
20	4	$2^2 3^2 4^2$	43	30	96	78	147
21	4	$2^2 3^2 4^2 5^2$	77	57	45	176	459
22	5	$2^3 4^1 3^1 2^1 4^1 3^1$	124	102	65	201	390
23	5	$2^3 3^1 4^1 2^2 3^1$	62	61	113	103	231
24	5	$3^1 4^1 2^1 4^1 5^2 2^2$	72	136	76	359	1228
25	5	$3^2 2^1 4^1 5^1 3^1 2^2$	147	147	191	263	630
26	5	$2^2 3^2 4^2 5^2$	1440	1440	1440	1440	1440
27	6	$2^2 3^2 4^2$	576	576	576	576	576

Tabela 4.16 - Experimento 2 - Análise de similaridade: distâncias euclidianas. Legenda: max = máxima distância euclidiana permitida.

	max	IPOG-F	*jenny*	IPO-TConfig	PICT
Execução 1	116,64	1508,57	1527,66	1126,74	1554,81
Execução 2	116,155	1466,64	1504,65	1158,58	1375,99
Execução 3	116,17	1544,13	1548,43	1232,76	1506,69

4.3 Quasiexperimento: Efetividade

4.3.1 Definição e Contexto

Esse quasiexperimento foi realizado com o objetivo de comparar a efetividade (capacidade de encontrar defeitos) entre os conjuntos de casos de teste gerados via versão 1.2 do algoritmo TTR e os conjuntos de teste dos quatro algoritmos/ferramentas já mencionados anteriormente. Para isso, foi utilizada a Técnica de Teste de Software Análise de Mutantes. Como já foi dito na seção 2.1, A Técnica de Teste Análise de Mutantes, além de ser uma eficiente técnica para a revelação de defeitos, também é uma poderosa técnica para avaliar o conjunto de testes utilizada durante o processo de teste de software.

Para essa experiência, foi selecionado um estudo de caso da área espacial, denominado *Payload Data Handling Computer* (SWPDC), que no fundo, trata-se de um produto de software, em que sua versão simplificada, escrita em linguagem de programação Java foi utilizada. Com base em alguns métodos selecionados desse estudo de caso, foram construídas instâncias de testes, onde cada um dos algoritmos/ferramentas avaliados gerou umo conjunto de testes correspondente. A partir daí, foi realizada uma Análise de Mutantes com cada um dos algoritmos/ferramentas, e o escore de mutação resultante serviu como base de comparação para verificar qual delas possuí efetividade igual ou diferente da versão 1.2 do TTR.

4.3.2 Estudo de Caso: Payload Data Handling Computer (SWPDC)

O estudo de caso escolhido para a realização do quasiexperimento apresentado nesse trabalho, é um produto de software da área espacial denominado *Payload Data Handling Computer* (SWPDC), desenvolvido no contexto do projeto de pesquisa Qualidade de Software Embarcado em Aplicações Espaciais (QSEE - *Quality of Space Application Embedded Software*) (SANTIAGO JÚNIOR et al., 2007), em que foi realizada uma experiência que contemplava o desenvolvimento de softwares embarcados em satélites de carga.

A Figura 4.10 mostra a arquitetura computacional definida para o projeto no qual o SWPDC foi desenvolvido: *On-Board Data Handling* (OBDH), *Payload Data Handling Computer* (PDC), *Event Pre-Processors* (EPPs; EPP H1 e EPP H2), e *Ionospheric Plasma Experiments* (IONEX). O computador OBDH é a plataforma do satélite responsável pelo processamento e *payload* de informações e formação/geração de dados para serem transmitidos a Estação Terrestre. Essas informações são obti-

Tabela 4.17 - Quasiexperimento - Hipóteses

Hypothesis	IPOG-F	jenny	IPO-TConfig	PICT
Hipótese Nula - Não existe diferença de efetividade entre o TTR e outra solução	H19.0	H20.0	H21.0	H22.0
Hipótese Alternativa - Existe diferença de efetividade entre o TTR e outra solução	H19.1	H20.1	H21.1	H22.1

das por meio de dois instrumentos científicos (note os dois retângulos pontilhados). Contudo, o principal instrumento é o que o PDC faz parte, uma vez que, o SWPDC está embarcado no PDC. O principal objetivo do PDC é obter dados científicos dos EPPs e transmití-los ao OBDH. EPPs são processadores de eventos responsáveis pelo rápido processamento de sinais de câmera de raio-X (SANTIAGO JÚNIOR, 2011).

Com isso, os principais objetivos do SWPDC são: (i) interagir com os EPPs, a fim de coletar dados científicos, de diagnóstico e de teste; (ii) formatação de dados; (iii) administração de memória para armazenar temporariamente os dados para serem transferidos ao OBDH; (iv) implementação de mecanismos de fluxo de controle; (v) geração de dados de limpeza; (vi) implementação de mecanismos complexos de tolerância de falhas; e (vii) carregar os novos programas durante o vôo.

Para esse quasiexperimento, foi utilizada uma versão simplificada do SWPDC, feita na linguagem de programação Java no contexto do projeto da FAPESP *VVTransv: Um Método para Teste e Verificação Formal Transversal ao Desenvolvimento de Sistemas Críticos* (MELO et al.,), onde foi selecionado alguns de seus métodos para realização da Análise de Mutantes.

4.3.3 Hipóteses e Variáveis

A Tabela 4.17 ilustra as hipóteses, e os seus números de identificação, para a avaliação do custo relacionado a efetividade dos conjuntos de teste geradas.

Com relação às variáveis envolvidas nesta experiência, destacam-se as variáveis independentes e variáveis dependentes, cujas definições já foram discutidas na seção 4.1.3. Para o primeiro tipo, foram identificados os algoritmos/ferramentas para gerar *designs* combinatóriais, o estudo de caso de teste considerado e a linguagem de programação em que os algoritmos foram implementados: Java (TTR, IPOG-F, IPO-

TConfig), C++ (PICT) e C (*jenny*). Para as variáveis dependentes, identificaram-se o número de casos de teste gerados, o resultado da Análise de Mutantes, e o resultado da análise de similaridade entre os conjuntos de testes.

4.3.4 Descrição do Experimento

Esse experimento foi conduzido de acordo com o objetivo de comparar a efetividade entre os conjuntos de casos de teste produzidos pelo TTR versão 1.2 e outras quatro soluções: IPOG-F, *jenny*, IPO-TConfig, and PICT. Para atingir tal objetivo, foi utilizado a técnica de Análise de Mutantes (DEMILLO et al., 1978; DELAMARO et al., 2007; MATHUR, 2008). Foi selecionado dois métodos de um produto de software da área espacial desenvolvido pelo Instituto Nacional de Pesquisas Espaciais (INPE). Essa aplicação é chamada de Software for the Payload Data Handling Computer (SWPDC) (SANTIAGO JÚNIOR et al., 2007; SANTIAGO JÚNIOR, 2011; SANTIAGO JÚNIOR; VIJAYKUMAR, 2012) criado em um projeto de pesquisa. Os métodos selecionados foram: (i) recuperar. Esse método recupera dados de um buffer de memória dado um indice; (ii) adicionar. Esse método adiciona dados em um buffer de memória onde o tipo do dado e e os próprios dados são os parâmetros.

A Análise de Mutantes avaliou o software SWPDC sob o ponto de vista do Teste de Unidade, portanto, foram selecionados alguns métodos relevantes desse estudo de caso. A Tabela 4.18 mostra os métodos selecionados, onde a primeira coluna corresponde as classes as quais esses métodos pertencem. Com base em cada um desses métodos, foram construídas instâncias de testes. Para isso, foi considerado que as variáveis internas e os parâmetros de cada método corresponderiam aos fatores das instâncias, e foi utilizado o critério de teste Particionamento em Classes de Equivalência[8] nessas variáveis para se obter os respectivos níveis, como mostra a Tabela 4.19.

Por exemplo, a variável *index* na Tabela 4.18 corresponde a um fator, e tem seu domínio particionado nas classes de equivalência da Tabela 4.19, portanto seus níveis são quatro: zero, um, algum número maior que um e algum número negativo. Portanto, para esse experimento, foram considerados 6 fatores, com respectivamente 4, 2, 3, 6, 3 e 4 níveis cada um (Observe na Tabela 4.19 que a quantidade de classes de equivalencia de cada fator é a mesma quantidade de níveis correspondente a esse fator).

[8]Particionamento em Classe de Equivalência, segundo (DELAMARO et al., 2007) é a divisão do domínio de entrada em classes, em que os integrantes de uma mesma classe são tratados da mesma forma. Dessa maneira, em teoria, basta selecionar um integrante de cada classe para fins de testes.

Tabela 4.18 - Quasiexperimento - Métodos do SWPDC selecionados para Análise de Mutantes.

Classe	Método	Atributos de Classe	Parâmetros	Retorno
GerenciarDados	adicionar	posBuffer	destino dado	posBuffer
BufferSimples	recuperar	dadoRecuperado m_ixwr	index	dadoRecuperado

Tabela 4.19 - Quasiexperimento - Particionamento em Classes de Equivalência das variáveis selecionadas para a Análise de Mutantes.

Fatores	Classes de Equivalência (níveis)
index	$= 0$ $= 1$ > 1 < 0
m_ixwr	> 0 $= 0$
dadoRecuperado	> 0 $= 0$ < 0
destino	tbRelatoEventos tbCientifico tbDiagnose tbHousekeeping tbTemperatura tbTeste
dado	> 0 $= 0$ < 0
posBuffer	$+1$ $+2$ $> +2$ noChange

Com isso, duas instâncias de teste foram construídas, considerando o valor do *strength = 2*, como pode-se observar na Tabela 4.20. Cada uma dessas instâncias foi submetida a cada um dos algoritmos/ferramentas considerados nesse experimento. A partir desse ponto, cada uma dos conjuntos geradas pelos algoritmos deu origem a um *script* de teste automatizado do JUnit (JUNIT, 2016). Em seguida, foram gerados mutantes a nível de método para as duas classes avaliadas por meio da ferramenta (OFFUTT et al., 2006), *BufferSimples* e *GerenciarDados*, considerando respectiva-

mente os métodos, *recuperar* para a primeira classe e *adicionar* para a segunda classe. Na Tabela 4.21 é apresentado a quantidade de mutantes a nível de método para cada um dos métodos avaliados. Com base na Análise de Mutantes, os escores de mutação foram calculados de acordo com a equação 2.1, e foram comparados ao escore obtido pelo algoritmo TTR 1.2, levando-se em consideração um intervalo de 10% de similaridade, ou seja, se algum outro algoritmo/ferramenta possuir um escore de mutação superior a 10%, implica que o TTR 1.2 e o algoritmo/ferramenta comparado não possuem a mesma efetividade para revelação de defeitos.

Tabela 4.20 - Quasiexperimento - Quantidade de casos de testes gerados por cada algoritmo/ferramenta para o quasiexperimento.

Método	Strength	Instâncias	*jenny*	TCONFIG	PICT	IPOG	TTR 1.2
adicionar	2	$4^1 2^1 3^1$	13	12	12	12	12
recuperar	2	$6^1 3^1 4^1$	24	24	24	24	24

Tabela 4.21 - Quasiexperimento - Quantidade de mutantes a nível de método gerados para cada método avaliado.

Método	Quantidade de mutantes gerados
recuperar	110
adicionar	33

Durante a execução do experimento, os dados foram gerados de forma automática, ou seja, os casos de teste correspondentes a Tabela 4.20. Com relação a geração dos mutantes, também foram gerados de forma automática de acordo com os operadosres de mutação selecionados. No entanto, devido a inconsistencia dos resultados gerados devido a diferentes sistemas operacionais utilizados, se tornou necessário que cada um deles fosse executado de forma manual, um por um. Além disso, a determinação dos mutantes equivalentes foi feita de forma manual e visual.

4.3.5 Validade

Nesta seção, será discutido alguns aspectos relacionados à validade do experimento. Quanto à validade da conclusão, é importante considerar a confiabilidade das métricas. Neste estudo, as quantidades de casos de teste e os mutantes foram obtidas de forma automática pelos algoritmos/ferramentas, e é possível obter os mesmos resultados, no caso de replicação deste experimento por outros pesquisadores.

As ameaças à validade interna comprometem a confiança ao afirmar que há uma relação entre variáveis dependentes e independentes. Não houve fatores que interferissem nessa relação porque os participantes, ou seja, instâncias de testes, foram engessadas, uma vez que foram selecionadas com base em parâmetros existentes em cada um dos métodos avaliados. Não houve eventos imprevistos para interromper a coleta de métricas: uma vez iniciada a geração de casos de teste seguiu rigorosamente os algoritmos implementados nas ferramentas, e o mesmo vale para a geração dos mutantes. Da mesma forma, a validade da construção também foi assegurada, uma vez que os algoritmos/ferramentas tradicionais de *designs* combinatóriais foram usados para serem comparados ao TTR 1.2.

Finalmente, as ameaças à validade externa comprometem a confiança em afirmar que os resultados do estudo podem ser generalizados para e entre indivíduos, contextos e sob a perspectiva temporal. Basicamente, pode-se dividir as ameaças à validade externa em duas categorias: ameaças à população e ameaças ecológicas.

As ameaças à população referem-se à significancia do conjunto de amostras da população utilizada no estudo. Por ser composto de apenas duas amostras, e não haver estudos parecidos com este, não podemos considerar significativa o conjunto de amostras a ponto de generalizar os resultados.

As ameaças ecológicas referem-se ao grau em que os resultados podem ser generalizados entre diferentes configurações. Os testes de interação (por exemplo, realizar um pré-teste com os participantes da experiência) e o efeito *Hawthorne* (devido aos participantes simplesmente sentirem-se estimulados por saber que participam de uma experiência inovadora) são alguns dos tipos desta ameaça. Os participantes dessa experiência são as instâncias de teste, portanto, este tipo de ameaça não se aplica a esse experimento.

4.3.6 Resultados

Considerando a efetividade, de acordo com os métodos descritos na Seção 4.3.4, as instâncias de teste foram geradas por cada um dos algoritmos/ferramentas avaliados. Cada uma dessas instâncias, foi transformada em um *script* para Teste de Unidade, considerando as classes e métodos (Veja Tabela 4.18) do estudo de caso considerado. Os mutantes gerados foram reúnidos na Tabela 4.22, onde também é possível verificar o valor do escore de mutação para cada caso.

A Tabela 4.22 corresponde a primeira abordagem de avaliação, onde é avaliado o

método *adicionar*, e a segunda abordagem de avaliação, onde é avaliado o método *recuperar*. Na grande maioria dos casos é possível verificar que, o valor do escore de mutação atingido por cada uma dos conjuntos de teste geradas pelos cinco algoritmos/ferramentas avaliados é igual.

A partir daí, foi realizada a comparação entre os escores de mutação produzidos paro conjunto de teste gerada por cada algoritmo/ferramenta, considerando o intervalo de 10% de diferença. Observe que, para a primeira abordagem de avaliação, a diferença entre o escore de mutação correspondente a cada um dos algoritmos/ferramentas é 0%, e para a segunda abordagem de avaliação tem-se: diferença de 2% nos escores de mutação gerados pelos conjuntos de teste produzidas pelo IPOG-F e *jenny* comparadas ao escore de mutação produzido pelo conjunto de teste gerada pelo TTR (sendo que o o escore de mutação produzido pelo TTR é maior), e diferença de 6% nos escores de mutação gerados pelos conjuntos de teste produzidas pelo PICT e IPO-TConfig comparadas ao escore de mutação produzido pelo conjunto de teste gerada pelo TTR (sendo que o o escore de mutação produzido pelo TTR é maior). Portanto, não há evidencias suficientes para a rejeição das hipóteses nulas de 19 a 22, uma vez que nenhuma das diferenças entre os escores de mutação do TTR versão 1.2 e dos demais algoritmos/ferramentas comparados: não há diferença de efetividade entre o algoritmo TTR versão 1.2 e os algoritmos/ferramentas IPOG-F, *jenny*, PICT e IPO-TConfig.

Tabela 4.22 - Quasiexperimento - Análise de Mutantes. Legenda: mt = mutantes totais; mm = mutantes mortos; mv = mutantes vivos; me = mutantes equivalentes; em = escore de mutanção (Veja equação 2.1).

Classe	Método	Algoritmo	mt	mm	mv	me	em
GerenciarDados	adicionar	IPOG-F	33	7	26	22	0,64
GerenciarDados	adicionar	*jenny*	33	7	26	22	0,64
GerenciarDados	adicionar	PICT	33	7	26	22	0,64
GerenciarDados	adicionar	IPO-TConfig	33	7	26	22	0,64
GerenciarDados	adicionar	TTR	33	7	26	22	0,64
BufferSimples	recuperar	IPOG-F	110	68	42	33	0,88
BufferSimples	recuperar	*jenny*	110	68	42	33	0,88
BufferSimples	recuperar	PICT	110	65	45	33	0,84
BufferSimples	recuperar	IPO-TConfig	110	65	45	33	0,84
BufferSimples	recuperar	TTR	110	69	41	33	0,9

4.4 Considerações Finais Sobre esse Capítulo

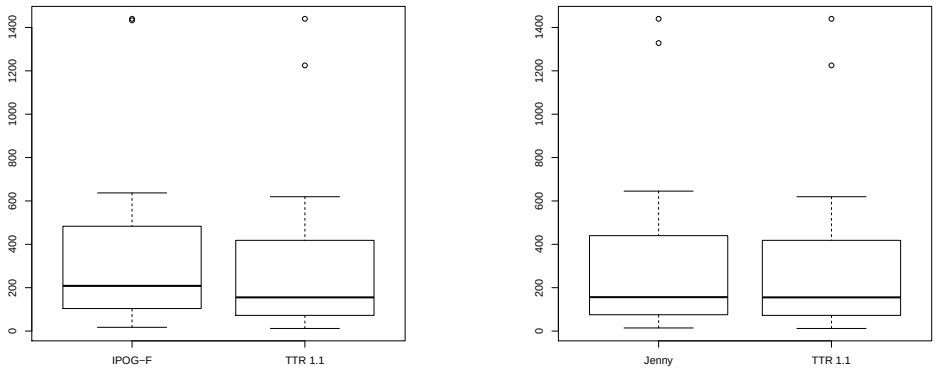
Esse capítulo apresentou dois experimentos controlados relacionados ao: (i) custo relacionado ao tamanho do conjunto de teste e tempo de geração das mesmas, e uma análise de similaridade entre os conjuntos de testes geradas pelos algoritmos/ferramentas IPOG-F, TConfig, *jenny* e PICT e a versão 1.1 do algoritmo TTR, e; (ii) custo relacionado ao tamanho do conjunto de testes, e análise de similaridade entre os conjuntos de teste geradas pela versão 1.2 do algoritmo TTR, e os algoritmos/ferramentas comparados. Os resultados do primeiro experimento mostram que, o TTR 1.1 foi o melhor em termos de quantidade de casos de teste gerados (conjuntos de testes menores), mas teve o desempenho mais fraco em termos de tempo para gerar os casos de teste. No entanto, o fato de que o TTR versão 1.1 gera conjuntos de teste menores, faz com que se utilize menos tempo para executar os conjuntos de testes, tornando o Processo de Teste como um todo menos demorado. A análise de similaridade mostra que o conjunto de casos de teste gerados via TTR versão 1.1 não são semelhantes aos criados pelas outras soluções. Assim, o TTR 1.1 tende a explorar diferentes comportamentos em comparação com os outros algoritmos/ferramentas avaliados.

Os resultados do segundo experimento controlado mostraram que o algoritmo proposto, TTR 1.2, mostrou-se a melhor opção, uma vez que fornece conjuntos de teste com a menor quantidade de casos de teste, comparado aos algoritmos/ferramentas avaliados. Sob o ponto de vista da análise de similaridade, os conjuntos de teste gerados pelo TTR 1.2 não são semelhantes a nenhum dos conjuntos de casos de teste gerados pelas outras abordagens. Isto pode ser uma indicação de que o TTR tende a estimular diferentes partes do SUT (Software Under Test) em relação aos outros algoritmos/ferramentas.

Adicionalmente, foi realizado um quasiexperimento em que foi avaliado a eficiencia dos conjuntos de teste geradas pelos algoritmos comparados, por meio da Técnica de Teste de Software Análise de Mutantes. Os resultados mostraram que todos os algoritmos possuem efetividade similar, no entanto, não é possível generalizar os resultados desse quasiexperimento, uma vez que a quantidade de amostras é pequena.

O próximo capítulo apresenta as conclusões e direções futuras dessa pesquisa.

Figura 4.6 - Experimento 1 - Gráficos BoxPlot dos dados em relação ao tempo para gerar os conjuntos de testes geradas para o TTR 1.1



(a) (b)

Figura 4.7 - Experimento 2 - Gráficos Q-QPLot dos dados do TTR 1.2: quantidade de casos de testes

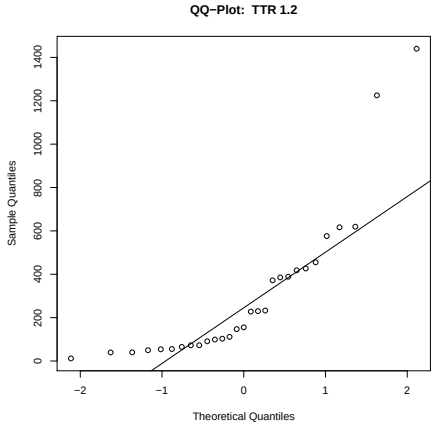


Figura 4.8 - Experimento 2 - Histograma dos dados do TTR 1.2: quantidade de casos de testes

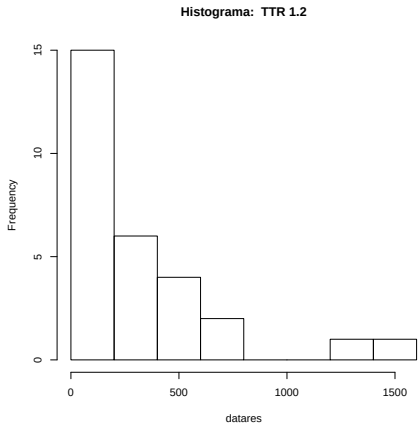
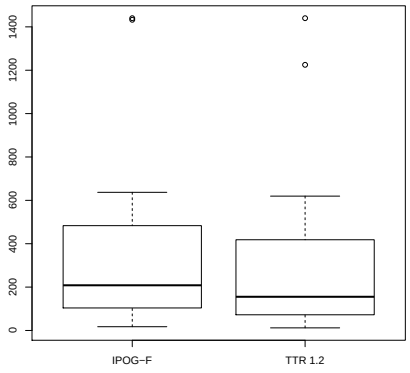
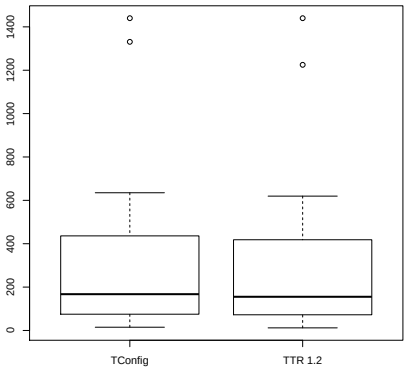


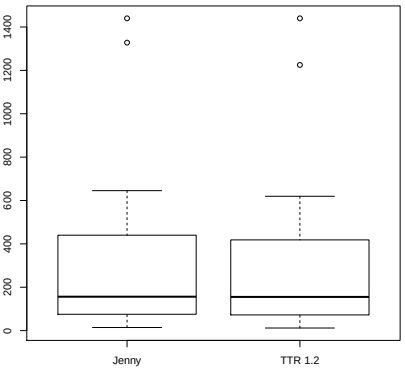
Figura 4.9 - Experimento 2 - Gráficos BoxPlot dos dados em relação ao tamanho daos conjuntos de testes gerados



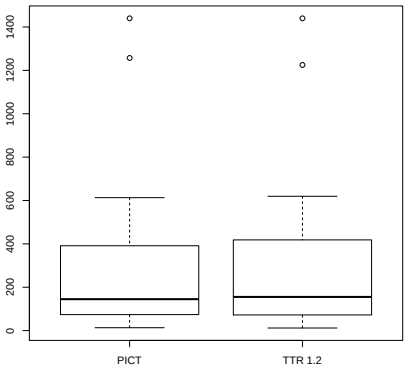
(a)



(b)

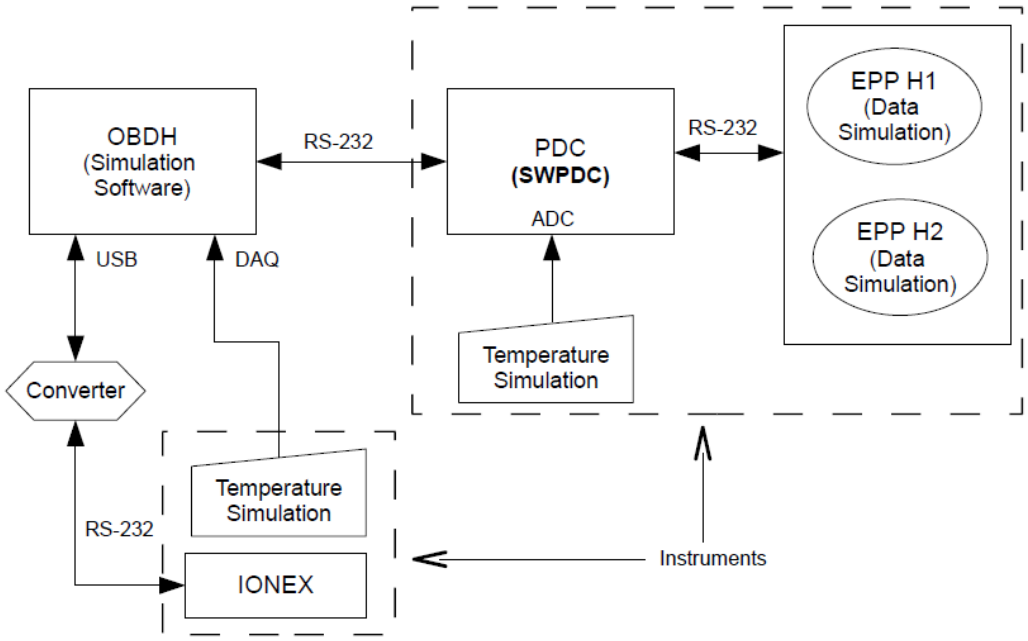


(c)



(d)

Figura 4.10 - Quasiexperimento - Arquitetura física definida para o projeto QSEE. Legenda: ADC = *Analog-to-Digital Converter*; DAQ = *Data Acquisition Board*; RS-232 = *Recommended Standard 232*; USB = *Universal Serial Bus*. Fonte: (SANTIAGO JÚNIOR, 2011).



5 CONCLUSÕES

A questão da qualidade é fundamental para qualquer tipo de produto, o que inclui produtos de software. Um dos processos que faz parte dos esforços da qualidade para produtos de sofware é, justamente, o processo de teste de software. E uma das principais atividades do processo de teste é a geração/seleção de casos de teste. Nesse contexto, é muito relevante dispor de métodos/técnicas que possibilitem a geração de um conjunto de casos de teste que demande menos esforço para ser executado e que, ao mesmo tempo, seja eficiente. Assim, *designs* combinatoriais aparecem como uma alternativa interessante para se gerar *suites* de teste menores, mas também eficientes.

O objetivo dessa dissertação de mestrado foi apresentar uma nova maneira de gerar conjuntos de casos de teste via *designs* combinatoriais, sendo que tais casos de teste deveriam ter custo menor e efetividade comparável à soluções já existentes na literatura. Para alcançar esse objetivo, um novo algoritmo para a geração de casos de teste, denominado TTR, usando *designs* combinatoriais e mais especificamente via a técnica MCNV, foi proposto e implementado em linguagem de programação Java. Foi apresentada uma revisão bibliográfica com as principais soluções usadas para gerar MCNVs, assim como as versões 1.0 (BALERA; SANTIAGO JÚNIOR, 2015), 1.1 (BALERA; SANTIAGO JÚNIOR, 2016) e 1.2 do algoritmo TTR.

Para verificar o desempenho do algoritmo TTR em face de outras soluções tradicionais existentes na literatura, foram realizados dois experimentos controlados e um quasiexperimento. No primeiro experimento controlado, a versão 1.1 do algoritmo TTR foi comparada a quatro algoritmos/ferramentas existentes na literatura, IPOG-F, IPO-TConfig, *jenny* e PICT, em termos de custo mensurado em quantidade de casos de casos de teste e tempo para a geração das *suites* de teste. Além disso, foi realizada uma análise de similaridade entre as *suites* de teste. No segundo experimento controlado, uma versão melhorada, 1.2, do algoritmo TTR foi comparada com as mesmas quatro soluções anteriores em termos de quantidade de casos de teste e foi realizada, também, uma análise de similaridade. Em termos de efetividade, um quasiexperimento foi realizado comparando o TTR 1.2 com as outras quatro soluções por meio de análise de mutantes e com um estudo de caso da área espacial.

Quanto ao primeiro experimento controlado, mostrou-se que o TTR 1.1 apresentou um melhor desempenho em relação a menor quantidade de casos de teste gerados em comparação com os outros quatro algoritmos/ferramentas. Isso é algo importante,

pois menos casos de teste para executar significa, em geral, menos tempo para executar o conjunto de casos de teste e, portanto, contribuíndo para uma diminuição do tempo relacionado ao processo de teste de software como um todo. Com respeito ao tempo para gerar as *suites* de teste, o algoritmo TTR 1.1 apresentou o pior desempenho, ao qual pode ser atribuído ao fato de que o TTR 1.1 ser implementado em linguagem de programação Java, que é considerada menos favorável em termos de aspectos de tempo real, devido ao fato de se tratar de uma linguagem interpretada. A análise de similaridade revelou que o algoritmo TTR 1.1 não gera *suites* de teste similar a nenhum dos outros quatro algoritmo/ferramentas comparados, o que indica que o TTR 1.1 pode exercitar partes diferentes do SST em relação aos outros algoritmos/ferramentas.

Com relação ao segundo experimento controlado, mostrou-se que o TTR 1.2 apresentou um melhor desempenho em relação a menor quantidade de casos de teste gerados em comparação com as outras quatro abordagens. A análise de similaridade para esse experimento também revelou que o algoritmo TTR 1.2 não gera *suites* de teste similar a nenhum dos outros quatro algoritmos/ferramentas comparados.

Quanto ao quasiexperimento para avaliar a efetividade (capacidade para revelação de defeitos) dos casos de teste produzidos por cada uma das soluções, por meio de análise de mutantes realizada com o software SWPDC, conclui-se que as 5 soluções avaliadas (IPOG-F, PICT, *jenny*, TConfig e TTR 1.2) possuem capacidade de revelação de defeitos equivalente. No entanto, em função da baixa quantidade de amostras envolvidas nesse experimento, não é possível generalizar os seus resultados.

Com relação as diferenças entre as três versões do algoritmo TTR, deve-se considerar que, na versão 1.0, a ordem de entrada em que os fatores e níveis são inseridos não era importante. Percebeu-se que relaxar essa decisão, não ordenando fatores, teve como resultado principal uma diminuição, ainda maior, em relação a quantidade de casos de teste gerados. Isso resultou na versão 1.1 do algoritmo. Embora a versão 1.1 tem sensibilidade à ordem de entrada de fatores, esse fato é compensado por um conjunto de casos de teste menor. Já a versão 1.2, também possui sensibilidade com relação a ordem dos fatores apresentados ao algoritmo, no entanto, tem como diferencial o fato de não necessitar que todas as *t-tuplas* a serem cobertas sejam geradas no início (não é mais necessário criar a matriz de *t-tuplas*, Θ). Além disso, a nova versão do algoritmo tende a gerar uma menor quantidade de casos de teste, em relação as versões anteriores. No entanto, o tempo para a geração das *suites* de teste, dessa nova versão 1.2, aumentou em relação as versões anteriores. Porém, o fato de

não gerar a matriz de t-tuplas, na versão 1.2, é interessante pois pode demandar menos memória para que a implementação do algoritmo seja executada.

Diante de todos os fatos apresentados, pode-se concluir que os resultados dessas três avaliações rigorosas são que o TTR foi o algoritmo que apresentou melhor custo (menor quantidade de casos de teste para serem executados), mas que não há diferença de efetividade entre o TTR e as demais soluções. Além disso, as *suites* de teste não são similares, comparando o TTR com as outras abordagens. Desse modo, pode-se afirmar que o TTR foi superior aos demais algoritmos/ferramentas pois teve mesma efetividade, porém melhor custo. Desse modo, acredita-se que o algoritmo TTR seja uma solução promissora para a geração de casos de teste de software via *designs* combinatoriais, com grande potencial para auxiliar o processo de teste de sistemas de software reais e complexos, como sistemas espaciais.

5.1 Trabalhos Futuros

Uma lista de iniciativas para dar continuidade a esse trabalho de pesquisa está apresentada a seguir:

- Realizar um experimento controlado, com mais amostras do que apresentado na Seção 4.3, relativo a efetividade do TTR comparado aos outros quatro algoritmos/ferramentas. Além disso, utilizar operadores de mutação do nível de classe, além daqueles do nível de método;

- Realizar um experimento controlado, com mais amostras do que apresentado na Seção 4.3, relativo a efetividade do TTR comparado aos outros quatro algoritmos/ferramentas, considerando o consumo de memória requerido pelas soluções;

- Gerar uma nova abordagem para a realocação das *t-tuplas* de maneira que torne possível a aplicação de técnicas de paralelização no algoritmo;

- Consideração de restrições[9]

- Migrar a implementação do algoritmo TTR para linguagens de programação como C/C++, uma vez que essas permitem melhor desempenho com relação ao tempo de execução da ferramenta implementada;

[9]Nesse caso, restrições se referem a requisitos a serem levados em conta durante a geração de casos de teste combinatoriais como, por exemplo, uma *t-tupla* que não pode ser considerada.

- Melhorar a interface humano/computador da implementação do algoritmo TTR;
- Realizar uma avaliação da usabilidade da ferramenta implementada junto a potenciais usuários da área aeroespacial.

REFERÊNCIAS BIBLIOGRÁFICAS

AHMED, B. S. Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing. **Engineering Science and Technology, an International Journal**, Elsevier, v. 19, n. 2, p. 737–753, 2016. 1

ANAND, S.; BURKE, E. K.; CHEN, T. Y.; CLARK, J.; COHEN, M. B.; GRIESKAMP, W.; HARMAN, M.; HARROLD, M. J.; MCMINN, P.; BERTOLINO, A.; LI, J. J.; ZHU, H. An orchestrated survey of methodologies for automated software test case generation. **Journal of Systems and Software**, v. 86, n. 8, p. 1978 – 2001, 2013. ISSN 0164-1212. Available from: <http://www.sciencedirect.com/science/article/pii/S0164121213000563>. 1

BALERA, J. M.; SANTIAGO JÚNIOR, V. A. T-tuple reallocation: An algorithm to create mixed-level covering arrays to support software test case generation. In: **15th International Conference on Computational Science and Its Applications (ICCSA)**. Berlin, Heidlberg: Springer International Publishing, 2015. p. 503–517. 3, 11, 21, 40, 75

BALERA, J. M.; SANTIAGO JÚNIOR, V. A. A. d. A controlled experiment for combinatorial testing. In: SIMPÓSIO BRASILEIRO DE TESTE DE SOFTWARE SISTEMÁTICO E AUTOMATIZADO, 1. (SAST), 19-23 set., Maringá, PR. **Proceedings...** [S.l.], 2016. p. 1–10. ISBN 9781450347662. Setores de Atividade: Pesquisa e desenvolvimento científico. Access in: 24 abr. 2017. 3, 11, 21, 40, 75

CAMPANHA, D. N.; SOUZA, S. R. S.; MALDONADO, J. C. Mutation testing in procedural and object-oriented paradigms: An evaluation of data structure programs. In: **2010 Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2010. p. 90–99. 7, 17, 18, 19, 42, 43

CAVALGNA, A.; GARGANTINI, A.; VAVASSORI, P. Combinatorial interaction testing with citlab. In: **Proceedings on 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. Nova York: IEEE, 2013. p. 376–382. 13, 15

CHOI, B. J.; DEMILLO, R. A.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P.; OFFUTT, A. J.; PAN, H.; SPAFFORD, E. H. The mothra tool set (software testing). In: **[1989] Proceedings of the Twenty-Second Annual**

Hawaii International Conference on System Sciences. Volume II: Software Track. [S.l.: s.n.], 1989. v. 2, p. 275–284 vol.2. 18

COHEN, D. M.; DALAL, S. R.; FREDMAN, M. L.; PATTON, G. C. The aetg system: An approach to testing based on combinatorial design. **IEEE Transactions on Software Engineering**, IEEE, Nova York, p. 43–66, 2014. 2, 12, 15

CZERWONKA, J. Pairwise testing in the real world: Practical extensions to test-case generators. In: **Proceedings 24th Pacific Northwest Software Quality Conference**. Portland: [s.n.], 2006. p. 285–294. 4, 5, 12, 15, 41, 47

DALAL, S. R.; JAIN, A.; KARUNANITHI, N.; LEATON, J. M.; LOTT, C. M.; PATTON, G. C.; HOROWITZ, B. Model-based testing in pratice. In: **Proceedings 21st International Conference on Software Engineering**. Nova York: AMC, 1999. p. 285–294. 2

DELAMARO, M. E. **Proteum - um ambiente de teste baseado na análise de mutantes**. PhD Thesis (PhD) — ICMC/USP, São Carlos, 1993. 7, 18, 19

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. Brasil: Elsevier, 2007. 1, 4, 7, 10, 15, 17, 66

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 11, n. 4, p. 34–41, apr. 1978. ISSN 0018-9162. Available from: <http://dx.doi.org/10.1109/C-M.1978.218136>. 7, 15, 17, 66

FORBES, M.; LAWRENCE, J.; LEI, Y.; KACKER, R. N.; KUHN, D. R. Refining the in-parameter-order strategy for constructing covering arrays. **Journal of Research of the National Institute of Standards and Technology**, p. 287–297, 2008. 2, 4, 5, 12, 13, 15, 41

HENARD, C.; PAPADAKIS, M.; PERROUIN, G.; KLEIN, J.; HEYMANS, P.; TRAON, Y. L. Bypassing the combinatorial explosion: Using similaruty to generate and prioritize t-wise test configurations for software product lines. **IEEE Transactions on Software Engineering**, p. 650–670, 2014. 13, 15

HERNANDEZ, L. G.; VALDEZ, N. R.; JIMENEZ, J. T. Construction of mixed covering arrays of variable strength using a tabu search approach. In: . Berlin,Heidelberg: Springer International Publisher, 2010. p. 51–64. 3, 12, 15

HIERONS, R. M.; BOGDANOV, K.; BOWEN, J. P.; CLEAVELAND, R.; DERRICK, J.; DICK, J.; GHEORGHE, M.; HARMAN, M.; KAPOOR, K.; KRAUSE, P.; LüTTGEN, G.; SIMONS, A. J. H.; VILKOMIR, S.; WOODWARD, M. R.; ZEDAN, H. Using formal specifications to support testing. **ACM Computing Surveys**, v. 41, n. 2, p. 1–76, 2009. 1

HUANG, C.-Y.; CHEN, C.-S.; LAI, C.-E. Evaluation and analysis of incorporating fuzzy expert system approach into test suite reduction. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 79, n. C, p. 79–105, nov. 2016. ISSN 0950-5849. Available from: <http://dx.doi.org/10.1016/j.infsof.2016.07.005>. 1

JENKINS, B. **Jenny: A Pairwise Tool**. 2016. Available from: <http://burtleburtle.net/bob/math/jenny.html>. Access in: 15 de Junho 2016. 3, 4, 5, 15, 41

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 37, n. 5, p. 649–678, sep. 2011. ISSN 0098-5589. Available from: <http://dx.doi.org/10.1109/TSE.2010.62>. 7, 17, 18

JUNIT. 2016. Available from: <http://junit.org/junit4/>. Access in: 29 de Dezembro 2016. 67

KHAN, S. U. R.; LEE, S. P.; AHMAD, R. W.; AKHUNZADA, A.; CHANG, V. A survey on test suite reduction frameworks and tools. **International Journal of Information Management**, v. 36, n. 6, Part A, p. 963 – 975, 2016. ISSN 0268-4012. Available from: <http://www.sciencedirect.com/science/article/pii/S0268401216303437>. 1

KOHL, M. **Introduction to Statistical Sata Analysis with R**. London: bookboon.com, 2015. 41, 45, 49, 53, 60

KUHN, D. R.; WALLACE, D. R.; GALLO, A. M. Software fault interactions and implications for software testing. **IEEE Trans. Software Eng.**, v. 30, n. 6, p. 418–421, 2004. Available from: <http://doi.ieeecomputersociety.org/10.1109/TSE.2004.24>. 2, 41

LEI, Y.; KACKER, R.; KUHN, D. R.; OKUN, V.; LAWRENCE, J. Ipog: A general strategy for t-way software testing. In: **Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)**. [S.l.: s.n.], 2007. p. 549–556. 2, 12, 13

LEMOS, O. A. L.; FERRARI, F. C.; ELER, M. M.; MALDONADO, J. C.; MASIEIRO, P. C. Evaluation studies of software testing research in Brazil and in the world: A survey of two premier software engineering conferences. **The Journal of Systems and Software**, v. 86, n. 4, p. 951–969, 2013. 4, 7, 18, 19, 41

MATHUR, A. P. **Foundations of Software Testing**. Delhi, India: Dorling Kindersley (India), Pearson Education in South Asia, 2008. 689 p. xiii, 1, 4, 7, 9, 10, 11, 66

MATTHES, E. **Curso Intensivo de Python**. Brazil: novatec, 2016. 41

MELO, A. C. V.; SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L. **VVTransv: um Método para Teste e Verificação Formal Transversal ao Desenvolvimento de Sistemas Críticos**. http://www.bv.fapesp.br/pt/auxilios/81907/vvtransv-um-metodo-para-teste-e-verificacao-formal-transversal-ao-desenvolvimento- Access in: 29 de Dezembro 2016. 65

MOHI-ALDEEN, S. M.; MOHAMAD, R.; DERIS, S. Application of negative selection algorithm (nsa) for test data generation of path testing. **Applied Soft Computing**, v. 49, p. 1118 – 1128, 2016. ISSN 1568-4946. Available from: <http://www.sciencedirect.com/science/article/pii/S1568494616305038>. 1

NETO, A. C. D. **Seleção de técnicas de teste baseado em modelos**. 2009. 220 p. PhD Thesis (Thesis (PhD in Computing and Systems Engineering)) — Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, RJ, Brazil, 2009. 1

NURMELA, K. J.; ÖSTERGåRD, P. R. J. **Constructing Covering Designs by Simulated Annealing**. 1993. 2

OFFUTT, J.; MA, Y.; KWON, Y. The class-level mutants of mujava. In: **Proceedings of the 2006 International Workshop on Automation of Software Test**. Shanghai, China: ACM, 2006. p. 78–84. ISBN 1-59593-408-1. 18, 19, 67

PETKE, J.; COHEN, M. B.; HARMAN, M.; YOO, S. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. **IEEE Transactions on Software Engineering**, v. 41, n. 9, p. 901–924, Sept 2015. ISSN 0098-5589. 2, 14, 15

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. Qualidade de software: Teoria e prática. Prentice Hall, Brasil, 2001. 9

ROTHERMEL, K. J.; COOK, C. R.; BURNETT, M. M.; SCHONFELD, J.; GREEN, T. R. G.; ROTHERMEL, G. Wysiwyt testing in the spreadsheet paradigm: an empirical evaluation. In: IEEE, 200. **Software Engineering, 2000. Proceedings of the 2000 International Conference on**. [S.l.], 2000. p. 230–239. ISBN 0270-5257. 7, 19

SAHIN, O.; AKAY, B. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. **Applied Soft Computing**, v. 49, p. 1202 – 1214, 2016. ISSN 1568-4946. Available from: <http://www.sciencedirect.com/science/article/pii/S156849461630504X>. 1

SANTIAGO JÚNIOR, V. A. **SOLIMVA: A METHODOLOGY FOR GENERATING MODEL-BASED TEST CASES FROM NATURAL LANGUAGE REQUIREMENTS AND DETECTING INCOMPLETENESS IN SOFTWARE SPECIFICATIONS**. PhD Thesis (PhD) — Instituto Nacional de Pesquisas Espaciais (INPE), 2011. xi, xiii, 4, 9, 10, 42, 65, 66, 74

SANTIAGO JÚNIOR, V. A.; MATTIELLO-FRANSCICO, F.; COSTA, R.; SILVA, W. P.; AMBROSIO, A. M. Qsee project: an experience in outsourcing software development for space applications. In: [**2007**] **Proceedings of International Conference on Software Engineering & Knowledge Engineering (SEKE)**. Boston, MA, USA: Skokie - USA: Knowledge Systems Institute Graduate School, 2007. v. 19, p. 51–56. 64, 66

SANTIAGO JÚNIOR, V. A.; SILVA, W. P.; VIJAYKUMAR, N. L. Shortening test case execution time for embedded software. In: IEEE INTERNATIONAL CONFERENCE ON SECURE SYSTEM INTEGRATION AND RELIABILITY IMPROVEMENT (SSIRI), 2., 2008, Yokohama, Japan. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p. 81–88. 1

SANTIAGO JÚNIOR, V. A.; VIJAYKUMAR, N. L. Generating model-based test cases from natural language requirements for space application software. **Software Quality Journal**, v. 20, n. 1, p. 77–143, 2012. DOI: 10.1007/s11219-011-9155-6. 4, 42, 66

SHAPIRO, S. S.; WILK, M. B. An analysis of variance test for normality (complete samples). **Biometrika**, v. 52, n. 3-4, p. 591–611, 1965. 45

STINSON, D. **Combinatorial Designs: Construction and Analysis**. Springer, 2004. ISBN 9780387954875. Available from: <https://books.google.com.br/books?id=jfF-rncVXNwC>. 2

TAI, K. C.; LEI, Y. A test generation strategy for pairwise testing. **IEEE Transactions on Software Engineering**, p. 109–111, 2002. 2

TATSUMI, K. Test case design support system. In: **Proceedings of the International Conference on Quality Control**. Tokio: [s.n.], 1987. p. 615–620. 2

WILLIAMS, A. W. Determination of test configurations for pair-wise interaction coverage. In: **International Conference on Testing Communicating Systems**. [S.l.: s.n.], 2000. p. 59–74. 4, 5, 41

WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. **Experimentation In Software Engineering: An Introduction**. USA: Kluwer Academic Publishers, 2000. 7, 19, 41

WONG, W. E.; MATHUR, A. P.; MALDONADO, J. C. Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In: LEE, M.; BARTA, B.-Z.; JULIFF, P. (Ed.). **Software Quality and Productivity: Theory, practice, education and training**. Boston, MA: Springer US, 1995. p. 258–265. ISBN 978-0-387-34848-3. Available from: <http://dx.doi.org/10.1007/978-0-387-34848-3_40>. 7, 17, 18

YILMAZ, C.; COHEN, M. B.; PORTER, A. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptative approach. **IEEE Transactions on Software Engineering**, p. 43–66, 2014. 14, 15

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: A survey. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., Chichester, UK, v. 22, n. 2, p. 67–120, mar. 2012. ISSN 0960-0833. Available from: <http://dx.doi.org/10.1002/stv.430>. 1

YU, L.; LEI, Y.; KACKER, R. N.; KUHN, D. R. Acst: A combinatorial test generation tool. In: **Proceedings on 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. Nova York: IEEE, 2013. p. 370–375. 13, 15

YU, L.; LEI, Y.; NOUROZBORAZJANY, M.; KACKER, R. N.; KUHN, D. R. An efficiente algorithm for constraint handling in combinatorial test generation. In:

2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. Nova York: IEEE, 2013. p. 242–251. 14

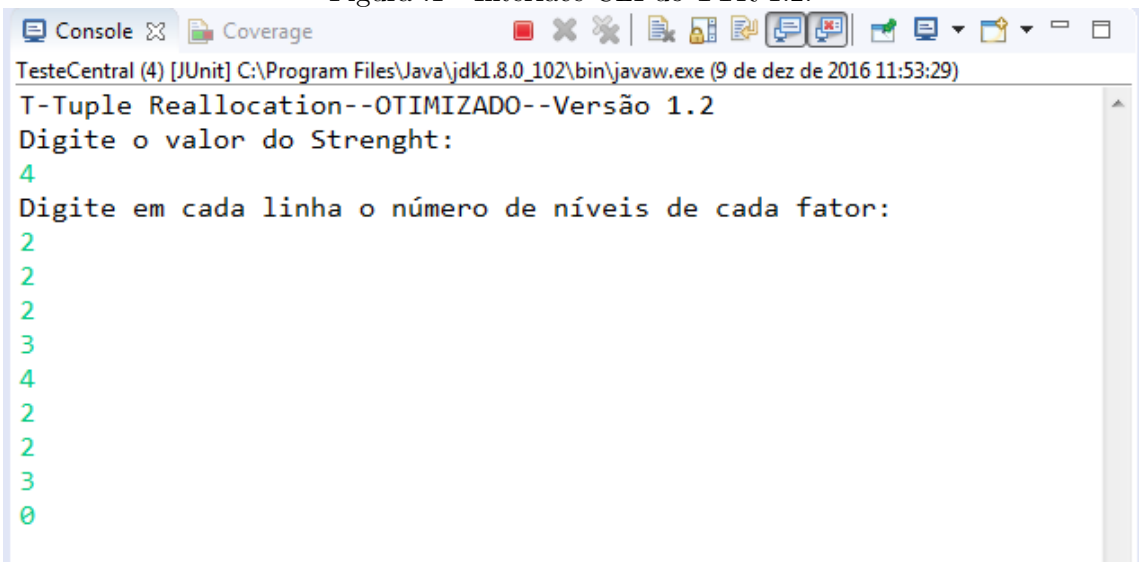
ZANNIER, C.; MELNIK, G.; MAURER, F. On the success of empirical studies in the international conference on software engineering. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: ACM, 2006. p. 341–350. ISBN 1-59593-375-1. 4, 18, 41

APÊNDICE B

.1 Tutorial para Utilização do TTR Versão 1.2

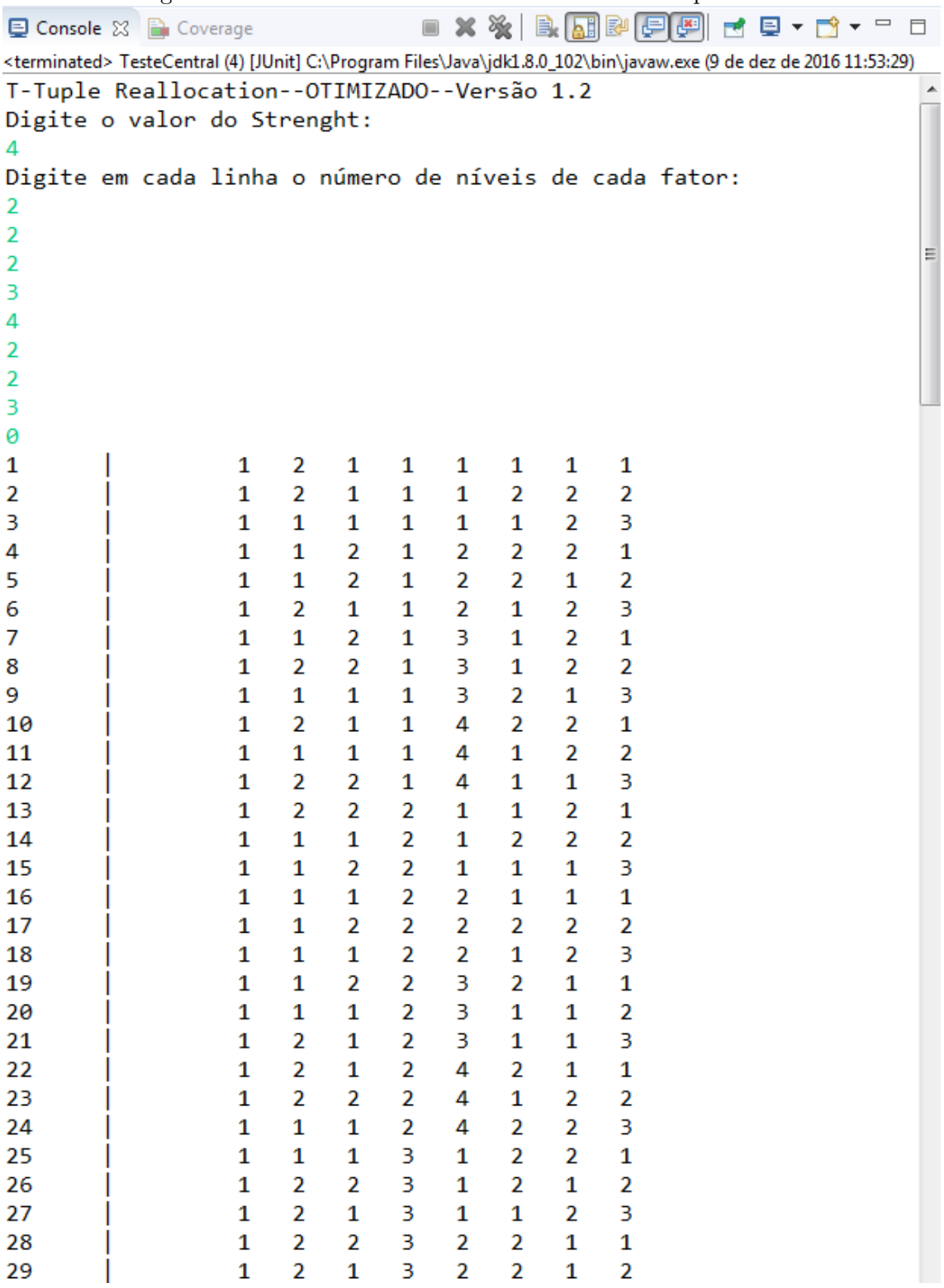
O Algoritmo TTR 1.2 não possuí Interface Gráfica, sendo utilizado por meio de linha de comando do próprio Ambiente de Desenvolvimento. A interação do usuário com o TTR é bem limitada: (i) o usuário é solicitado a digitar o valor do *Strength*; (ii) o usuário digita a quantidade de níveis de cada um dos fatores (cada linha repsresenta um fator); (iii) após digitar toda a sequencia de fatores, o usuário digita 0 para o TTR construir a *suite* de testes. Após a realização de todas as realocações de *t-tuplas*, o algoritmo exibe a *suite* de testes.

Figura .1 - Interface CLI do TTR 1.2.



Por exemplo, a Figura .1, o usuário digita o valor de *strenght* igual a 4, em seguida, digita o primeiro fator, com 2 níveis, em seguida o segundo fator, com 2 níveis, e assim por diante. Até que quando termina de digitar todos os fatores, é digitado 0. O resultado é mostrado na Figura .2, veja que a *suite* de teste é exibida, e cada uma de suas linhas representa um teste, que está numerado.

Figura .2 - Interface CLI do TTR 1.2: *suite* de teste produzida.

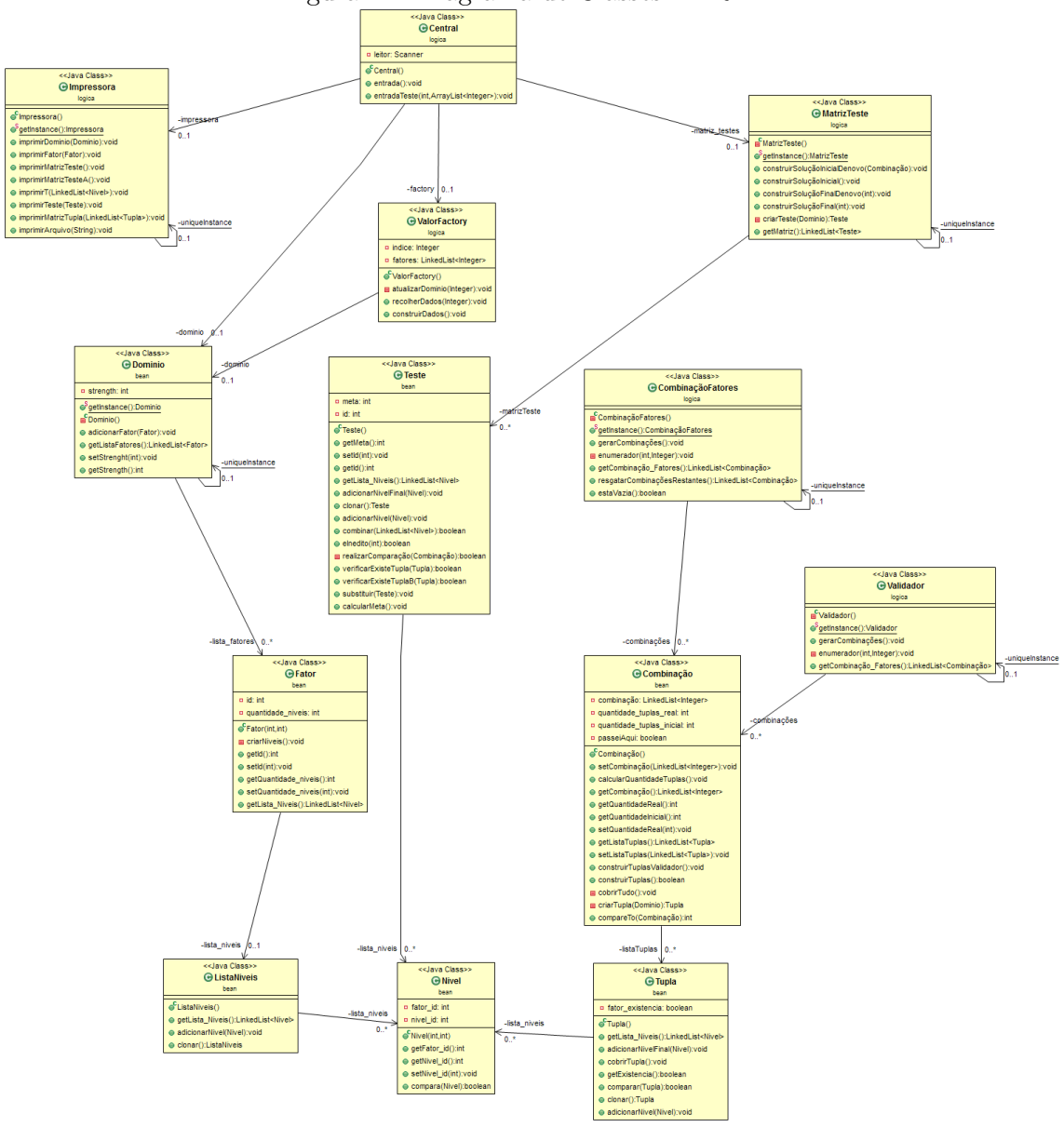


APÊNDICE A

.1 Diagrama de Classes da Implementação do Algoritmo TTR

A seguir, o Diagrama de Classes da versão 1.2 do algoritmo TTR. Observe que o algoritmo foi desenvolvido sob o paradigma de programação orientado a objetos. Onde as classes foram criadas para representar objetos do mundo real, como fatores, níveis, domínio, etc, que em sua grande maioria, são coleções de itens que são armazenadas em listas ligadas.

Figura .1 - Diagrama de Classes TTR 1.2.



PUBLICAÇÕES TÉCNICO-CIENTÍFICAS EDITADAS PELO INPE

Teses e Dissertações (TDI)

Teses e Dissertações apresentadas nos Cursos de Pós-Graduação do INPE.

Manuais Técnicos (MAN)

São publicações de caráter técnico que incluem normas, procedimentos, instruções e orientações.

Notas Técnico-Científicas (NTC)

Incluem resultados preliminares de pesquisa, descrição de equipamentos, descrição e ou documentação de programas de computador, descrição de sistemas e experimentos, apresentação de testes, dados, atlas, e documentação de projetos de engenharia.

Relatórios de Pesquisa (RPQ)

Reportam resultados ou progressos de pesquisas tanto de natureza técnica quanto científica, cujo nível seja compatível com o de uma publicação em periódico nacional ou internacional.

Propostas e Relatórios de Projetos (PRP)

São propostas de projetos técnico-científicos e relatórios de acompanhamento de projetos, atividades e convênios.

Publicações Didáticas (PUD)

Incluem apostilas, notas de aula e manuais didáticos.

Publicações Seriadas

São os seriados técnico-científicos: boletins, periódicos, anuários e anais de eventos (simpósios e congressos). Constam destas publicações o Internacional Standard Serial Number (ISSN), que é um código único e definitivo para identificação de títulos de seriados.

Programas de Computador (PDC)

São a seqüência de instruções ou códigos, expressos em uma linguagem de programação compilada ou interpretada, a ser executada por um computador para alcançar um determinado objetivo. Aceitam-se tanto programas fonte quanto os executáveis.

Pré-publicações (PRE)

Todos os artigos publicados em periódicos, anais e como capítulos de livros.