LUCILIA YOSHIE ARAKI

Um Algoritmo Evolutivo de Geração de Dados de Teste para Satisfazer Critérios Baseados em Código Objeto Java

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Profa. Dra. Silvia Regina Vergilio.

CURITIBA

2009

LUCILIA YOSHIE ARAKI

Um Algoritmo Evolutivo de Geração de Dados de Teste para Satisfazer Critérios Baseados em Código Objeto Java

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Profa. Dra. Silvia Regina Vergilio.

CURITIBA

2009

Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, da aluna Lucilia Yoshie Araki, avaliamos o trabalho intitulado, *"UM ALGORITMO EVOLUTIVO DE GERAÇÃO DE DADOS DE TESTE PARA SATISFAZER CRITÉRIOS EM CÓDIGO OBJETO JAVA"*, cuja defesa foi realizada no dia 27 de agosto de 2009, às 14:00 horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata.

Curitiba, 27 de agosto de 2009.

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR – Orientadora

Prof. Dr. Auri Marcelo Rizzo Vincenzi
UFG – Membro Externo

Profª. Dra. Aurora Ramirez Trinidad Pozo
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

Aos meus pais Cecilia e Kiyoshi,
ao meu irmão Luciano e
ao meu namorado Marcel.

AGRADECIMENTOS

Agradeço a minha orientadora, professora Dra. Silvia Regina Vergilio, por sua paciência e orientação nesta dissertação, bem como por todo o conhecimento por ela me transmitido.

Agradeço, também, a meus familiares e amigos, estando eles fisicamente próximos ou não, mas que me apóiam, colaboram com este trabalho e/ou ajudam-me a enfrentar esta jornada.

Meus sinceros agradecimentos, também, a todos meus professores, pois não seria possível realizar este empreendimento sem os conhecimentos por eles me fornecidos.

Gostaria de agradecer, também, ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro.

Por último, mas não menos importante, agradeço aos membros da banca, Dr. Auri Marcelo Rizzo Vincenzi e Dra. Aurora Trinidad Ramirez Pozo, por todo o tempo dispendido, sugestões e críticas apontadas para este trabalho.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AG	Algoritmo Genético
AGI	Algoritmo Genético Ineditismo
AM	Aprendizado de Máquina
CE	Computação Evolucionária
CUT	*Class Under Test* (Classe sendo testada)
EE	Estratégias de Evolução
IA	Inteligência Artificial
IC	Inteligência Computacional
JaBUTi	*Java Bytecode Understanding and Testing*
MEF	Máquina de Estados Finitos
MUT	*Method Under Test* (Método sendo Testado)
OO	Orientado a Objetos
PE	Programação Evolutiva
PG	Programação Genética
POKE-TOOL	*Potential Uses Criteria Tool for Program Testing*
POO	Programação Orientada a Objetos
PROTEUM	*Program Testing Using Mutants*
SC	Sistemas de Classificação
TDSGen/OO	*Test Data Set Generator for OO Software*

RESUMO

O paradigma de orientação a objetos ganhou bastante importância nas últimas décadas e garantir a qualidade de software desenvolvido nesse contexto ainda é um desafio. Com esse objetivo, e a fim de garantir a qualidade dos testes utilizados, a aplicação de um critério de teste é fundamental. Existem diversas ferramentas de suporte que permitem a utilização prática desses critérios, destacando-se a ferramenta JaBUTi que implementa critérios baseados em código objeto Java (*bytecode*), viabilizando o teste estrutural sem que o código fonte esteja disponível. Isso é bastante comum no caso da maioria dos componentes de *software*. Entretanto, poucos esforços ainda têm sido dedicados à tarefa de geração de dados de teste para esse contexto. Essa tarefa possui muitas limitações para ser completamente automatizada e geralmente consome muito esforço, elevando os custos de utilização das ferramentas e dos critérios de teste. Técnicas de Aprendizado de Máquina e Computação Evolutiva têm sido aplicadas com bastante sucesso para geração de dados de teste para *software* procedural, destacando-se entre estas, os Algoritmos Genéticos (AG). Recentemente alguns autores têm explorado o uso dessas técnicas também no contexto de *software* orientado a objetos. Entretanto, os trabalhos exploram poucos critérios, na maioria das vezes, somente critérios baseados em fluxo de controle e não estão integrados a uma ferramenta. Para permitir a geração de dados para satisfazer critérios baseados em *bytecode* no contexto de software orientado a objetos, esse trabalho propõe a utilização de Algoritmos Genéticos e considera aspectos particulares da atividade de teste. Os algoritmos são implementados e integrados à ferramenta JaBUTi. Além disso, os algoritmos são avaliados experimentalmente e os resultados mostram que eles auxiliam a tarefa de geração de dados de teste reduzindo custo e esforço gastos nesta atividade.

ABSTRACT

In the last years, the object-oriented paradigm has been intensively used and gained importance. However, to ensure the quality of object-oriented software is considered by some authors a challenge, and because of this, the use of a test criterion in this context is fundamental. There are some tools that support the application of such criteria, among them JaBUTi, which implements test criteria for Java bytecode. These criteria allow the test of components even if the source code is not available. This is very common in the testing of most components. However, few works address the test generation problem in the Object Oriented context. This task has many limitations to be completely automated, and this phase increases testing effort and costs. To overcome these limitations Machine Learning techniques, such as Genetic Algorithms, have been successfully explored for test generation in the context of procedural programs and most recently in the Oriented Object context. However, the existent works explore a low number of criteria, frequently only control-flow based criteria, such as all-nodes and all-edges. In addition to this, the algorithms are not integrated with a test tool. Because of this, this work explores the use of Genetic Algorithms for test data generation to satisfy the criteria implemented by JaBUTi, based on Java bytecode. The implemented algorithms consider particular aspects of the testing activity. Besides this, they were experimentally evaluated and the results show that they help in the generation of test data, reducing costs and effort spent in this task.

1 INTRODUÇÃO

1.1 Contexto

O paradigma de orientação a objetos surgiu com o objetivo de permitir um isolamento (encapsulamento) dos dados da forma como eles são manipulados. Essas e outras características fizeram com que o paradigma orientado a objetos se tornasse bastante popular em detrimento do paradigma procedimental, permitindo o desenvolvimento de soluções reusáveis e tornando-se a base para a criação de componentes de software (VINCENZI et al., 2007).

O uso de novos paradigmas que facilitem o desenvolvimento de programas é muito importante. Entretanto, isso não impede que alguns enganos sejam cometidos pelos programadores, refletindo em defeitos e possivelmente falhas de utilização. Além disso, suas novas características também introduzem novos e diferentes tipos de defeitos. Por causa disso, para garantir a qualidade do software em desenvolvimento, é fundamental a adaptação de técnicas e ferramentas de software existentes para esses novos contextos.

O teste de software é uma atividade essencial entre as de garantia de qualidade e a aplicação de um critério de teste é muito importante para assegurar a qualidade dos casos de teste utilizados. Eles auxiliam a escolha das melhores entradas (dados de teste) dentre um conjunto geralmente infinito e impraticável. Um critério de teste é um predicado a ser satisfeito que estabelece certos requisitos de teste, chamados elementos requeridos, que devem ser exercitados durante a execução do programa, ou seja, executados por um dado de teste (RAPPS e WEYUKER, 1985). A idéia é gerar os melhores dados que possam revelar a maioria dos defeitos com baixo custo (MALDONADO, 1991). O número de elementos exercitados fornece uma medida de cobertura que pode ser utilizada para avaliar o conjunto de dados de teste e considerar a atividade de teste encerrada.

Os critérios de teste são classificados de acordo com a informação utilizada para derivar os requisitos de teste e estabelecidos a partir de três técnicas básicas: *estrutural*, que utiliza a estrutura interna do programa para derivar os dados de teste; *funcional*, que deriva os dados de teste baseando-se nos requisitos funcionais do software; e, *baseada em defeitos*, que

deriva os dados de teste baseando-se em defeitos comuns cometidos durante o desenvolvimento de software.

O paradigma de orientação a objetos ganhou bastante importância nas últimas décadas, mas apesar disso, garantir a qualidade de software desenvolvido nesse contexto ainda é um desafio. A aplicação de um critério de teste é fundamental a fim de garantir a qualidade dos testes utilizados. Muitos são os trabalhos que introduzem critérios de teste considerando a Programação Orientada a Objetos (POO), baseados geralmente na especificação ou no programa. Os primeiros (BINDER, 1999; CHEN et al., 1998; HOFFMAN e STOOPER, 1997; OFFUT e IRVINE, 1995) consideram diversos diagramas associados a software orientado a objetos para derivar os requisitos de teste tais como os diagramas de estado que modelam comportamento do objeto, diagramas de caso de uso e etc. Uma dificuldade com esses critérios é a automatização e a especificação que pode ser de baixa qualidade e muito informal. Dentre os critérios baseados em programa, destacam-se os estruturais, baseados em fluxo de controle e dados (HAROLD e HOTERMEL, 1994; VINCENZI et al., 2006), e o critério baseado em mutação (BIEMAN et al., 2001). Os primeiros são geralmente baseados em grafos de fluxo de controle e nas interações de fluxo de dados que acontecem em uma dada sequência de chamadas de métodos de uma classe.

Existem diversas ferramentas de suporte que permitem a utilização prática de um critério de teste, destacando-se a ferramenta JaBUTi (VINCENZI et al., 2003). A JaBUTi implementa critérios baseado em código objeto (*bytecode*) Java. A vantagem desses critérios é que eles permitem o teste sem que o código fonte esteja disponível. Isso é bastante comum no caso de componentes.

As ferramentas, na maioria das vezes geram os elementos requeridos pelos critérios e realizam a análise da adequação de um conjunto de casos de teste com relação aos critérios implementados. No entanto, cabe ao usuário escolher um dado de teste que exercite os elementos requeridos. Embora a automatização dessa tarefa seja desejável, não existe um algoritmo de propósito geral para determinar um conjunto de casos de teste que satisfaça um dado critério. Não é possível nem mesmo determinar se esse conjunto existe. Isso torna o problema de geração de dados de teste indecidível possuindo muitas limitações tais como a identificação de caminhos não executáveis no programa (FRANKL, 1987) e a identificação de programas equivalentes (BUDD, 1981), etc.

Apesar dessas limitações, diferentes categorias de estratégias que podem ser utilizadas para gerar dados de teste para os diversos critérios são encontradas na literatura (VERGILIO et al., 2007). As mais conhecidas são geração aleatória de dados, geração com execução

simbólica, geração com execução dinâmica e, mais recentemente, geração utilizando-se algoritmos meta-heurísticos, que utilizam de técnicas de Computação Evolutiva e de Aprendizado de Máquina (AM) (VERGILIO et al., 2007). Essa última estratégia originou uma nova área de pesquisa chamada de Teste Evolutivo (WEGENER et al., 2001). O Teste Evolutivo tem ganhado bastante espaço, pois permite reduzir as limitações mencionadas acima. A maioria desses trabalhos utiliza Algoritmos Genéticos (AG).

Algoritmos Genéticos são métodos de busca e otimização que simulam os processos naturais de evolução (HOLLAND, 1975). Os indivíduos mais adaptados de uma população têm maior probabilidade de sobreviver e de gerar descendentes. A população geralmente representa um conjunto de dados de teste e evolui segundo uma função de aptidão (*fitness*) que geralmente é avaliada com base na cobertura de cada dado de teste para um critério determinado, ou de acordo com um dado objetivo que é a cobertura de um elemento requerido em particular. Na literatura encontram-se inúmeros trabalhos que utilizam AG para gerar dados para satisfazer diferentes critérios, por exemplo, para critérios estruturais (FERREIRA e VERGILIO, 2005; JONES et al, 1996; MICHAEL et al, 2001; ROPER et al, 1995; TRACEY et al, 1998; WEICHSELBAUM, 1998 e WEGENER et al, 2001) e baseados em defeitos (BOTTACI, 2001; CHUNG, 1998; FERREIRA e VERGILIO, 2005). Entretanto percebe-se que os critérios abordados são para programas tradicionais.

Recentemente, alguns autores voltaram-se para a geração de dados de teste evolucionário no contexto de POO. Esses trabalhos utilizam diferentes técnicas de Aprendizado de Máquina e com diferentes objetivos (SAGARNA et al, 2007; SEESING e GROSS, 2006; TONELLA, 2004; WAPPLER e LAMMERMANN, 2005; WAPPLER e WEGNER, 2006). Alguns destes trabalhos não possuem o objetivo de satisfazer um critério de teste. Os que possuem tal objetivo abordam apenas critérios baseados em fluxo de controle, quase sempre o critério todos-nós ou todos-ramos. Eles não consideram critérios baseados em *bytecode* e, além disso, não oferecem uma implementação do algoritmo integrada com uma ferramenta de suporte ao critério escolhido.

1.2 Justificativa

Dado o contexto acima que reflete o atual estado da arte no problema de geração de dados de teste para a satisfação de critérios de teste para software orientado a objetos, têm-se os seguintes pontos, que são motivações e justificam o presente trabalho:

- A importância de se aplicar um critério de teste, pois ele oferece medidas para quantificar a atividade de teste e para avaliar a qualidade do teste sendo efetuado;
- A existência de uma demanda por técnicas e ferramentas de geração de dados de teste para apoiar a utilização de critérios e ferramentas de teste no contexto de POO. Dentre esses critérios destacam-se os que utilizam diretamente o código objeto;
- A dificuldade para se automatizar completamente a tarefa de geração de dados de teste que possui limitações inerentes à própria atividade de teste e, os bons resultados obtidos com a utilização de algoritmos meta-heurísticos destacando-se os AG no teste evolutivo de programas tradicionais;
- A existência de poucos trabalhos para o teste evolutivo no contexto de POO. Vale a pena ressaltar que esses trabalhos consideram apenas poucos critérios de teste, na maioria somente critérios baseados em fluxo de controle, que não consideram o *bytecode*. Além disso, a maioria deles não se preocupa em oferecer um ambiente integrado a uma ferramenta;
- A integração com uma ferramenta e o suporte a diferentes critérios de teste é fundamental para permitir a aplicação de uma estratégia de teste que possibilite a utilização desses critérios de uma maneira complementar. Além disso, os critérios baseados em *bytecode* podem ser aplicados mesmo quando o código-fonte não estiver disponível, caso que ocorre frequentemente no teste da maioria dos componentes.

1.3 Objetivo

O presente trabalho tem como objetivo explorar o uso de algoritmos evolutivos, mais particularmente Algoritmos Genéticos, para a geração de dados de teste no contexto de Programação Orientada a Objetos (POO).

Para isso, uma ferramenta chamada TDSGen/OO foi implementada a fim de gerar dados de teste para satisfazer os critérios baseados em fluxo de controle e de dados implementados pela ferramenta JaBUTi, que extrai os requisitos a partir de código objeto (*bytecode*) Java. A idéia é oferecer um ambiente para a aplicação de todos os critérios implementados em uma estratégia de teste e assim, reduzir os custos e esforço gasto para aplicação dos critérios e, consequentemente, facilitar a atividade de teste de software desenvolvido com esse paradigma.

A ferramenta está baseada em AGs que consideram particularidades da atividade de teste e foi utilizada em um experimento de validação.

1.4 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma: nos Capítulos 2 e 3 são introduzidos, respectivamente, conceitos básicos sobre Algoritmos Genéticos e teste de software.

No Capítulo 4 revisa-se o teste de software orientado a objetos, relacionando esta dissertação com outros trabalhos similares.

No Capítulo 5 descreve-se a proposta para desenvolvimento de uma ferramenta que auxilie na geração de dados de teste e assim, reduza os custos e esforço gastos, facilitando a atividade de teste.

No Capítulo 6 encontram-se os resultados dos experimentos realizados.

O Capítulo 7 apresenta as conclusões e trabalhos futuros.

O trabalho contém 3 apêndices. O Apêndice A contém exemplos de relatórios gerados pela ferramenta JaBUTi.

O Apêndice B apresenta os comandos da JaBUTi utilizados.

No Apêndice C encontram-se os resultados para cada programa e cada estratégia.

2 ALGORITMOS GENÉTICOS

A Computação Evolutiva (CE) é uma ramificação da ciência da computação que, juntamente com o estudo de Sistemas *Fuzzy* e Redes Neurais, forma uma área denominada Inteligência Computacional (IC) (PALAZZO e CASTILHO, 1997). A computação evolutiva é constituída em princípio por cinco diferentes linhas: Algoritmos Genéticos, Programação Evolutiva (PE), Estratégias de Evolução ou Evolutivas (EE), Sistemas de Classificação ou Classificadores (SC) e Programação Genética (PG). A Figura 2.1 esquematiza esta classificação.

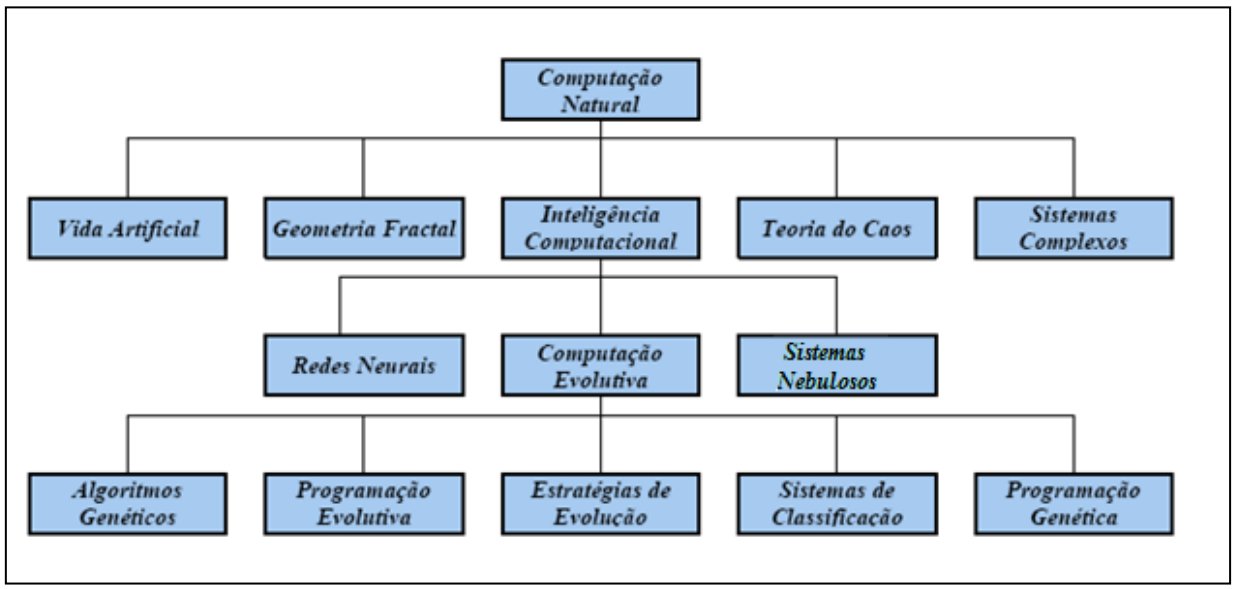


Figura 2.1: Taxonomia da Computação Natural (NAVAUX e SHUBEITA, 2003).

Ao longo do tempo, as metodologias que constituem os procedimentos do tipo evolutivo foram desenvolvidas independentemente umas das outras. O termo Algoritmos Evolutivos foi proposto em 1990, buscando reunir todas as variantes e também todas as outras técnicas que fossem baseadas em princípios evolutivos para a solução de problemas.

Os Algoritmos Evolutivos referem-se aos procedimentos baseados em princípios que são encontrados na evolução dos sistemas biológicos, tentando abstrair e imitar alguns dos mecanismos evolutivos para a resolução de problemas que requerem adaptação, busca e otimização. Usando esses princípios, a tarefa de encontrar a solução de um problema pode ser vista como uma tarefa de sobrevivência.

A principal vantagem dos Algoritmos Evolutivos é que eles possuem grande flexibilidade e adaptabilidade aos problemas reais. Estas técnicas trabalham com a modelagem real do problema, sem necessidade de informações auxiliares.

Dentre os algoritmos evolutivos existentes os Algoritmos Genéticos (AG) são de particular interesse para esse trabalho e são descritos neste capítulo.

2.1 Algoritmos Genéticos

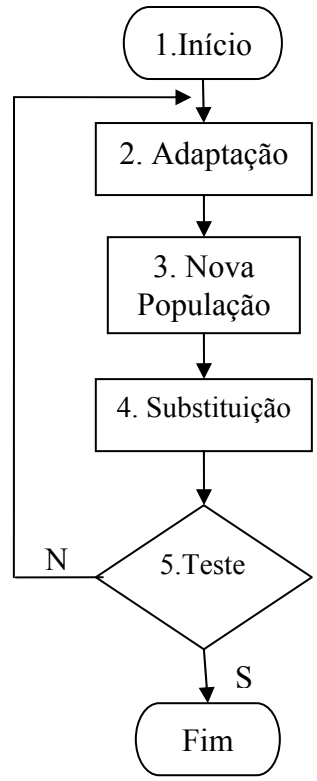
Algoritmos Genéticos (AGs) são algoritmos de busca baseados em mecanismos de seleção natural e da genética (GOLDBERG, 1989), que exploram a idéia da sobrevivência dos indivíduos mais aptos e do cruzamento de populações para criar novas estratégias de pesquisa.

John Holland (HOLLAND, 1975) acreditava que, usando a evolução, era possível produzir uma técnica para solucionar problemas difíceis da mesma forma que a natureza fazia para resolver os seus problemas, incorporando adequadamente as características da evolução natural a algoritmos computacionais.

Baseado nesta idéia, Holland iniciou a uma pesquisa sobre algoritmos que manipulavam "*strings*" de 0's (zeros) e 1's (uns), a qual ele deu o nome de cromossomos. Assim como na natureza, estes cromossomos não têm conhecimento nenhum sobre o tipo de problema que estão resolvendo. A única informação que eles dispõem é uma avaliação de cada cromossomo produzido para verificar quais os cromossomos que estão mais adaptados e, com base nisto, aumentar as suas chances de serem selecionados para a reprodução.

Segundo Concilio (2000), um Algoritmo Genético deve conter os seguintes componentes:

- uma representação genética para soluções candidatas ou potenciais (processo de codificação);
- uma população inicial, geralmente gerada aleatoriamente;
- função de avaliação (ou adaptação) classificando as soluções de acordo com sua adaptação ao meio (*fitness*);
- operadores genéticos;
- valores para os parâmetros genéticos (tamanho da população, probabilidades de aplicação dos operadores, entre outros). A Figura 2.2 mostra o funcionamento (Diagrama de Blocos) de um Algoritmo Genético.



1. Geração de população inicial, aleatoriamente.
2. É calculado o grau de adaptação de cada indivíduo (*fitness*).
3. É criada uma nova população através da aplicação dos operadores genéticos: Seleção Cruzamento e Mutação, de acordo com probabilidades definidas no algoritmo.
4. Substituição da população antiga pela população gerada.
5. Se condição inicial foi satisfeita, o algoritmo termina retornando a melhor solução encontrada.

Figura 2.2: Funcionamento de um Algoritmo Genético (adaptado de COSTA e BRUNA, 2002).

A aplicação de qualquer algoritmo evolutivo tem como aspectos essenciais a escolha da forma de representação dos cromossomos, a forma de criação da população inicial, a formulação da função de avaliação da adaptação (*fitness*), e os processos de seleção e geração de novos indivíduos (reprodução e mutação).

2.1.1 Indivíduo

Nos AGs o indivíduo é uma representação do espaço de busca do problema a ser resolvido, em geral na forma de sequências de bits ou caracteres. Nem sempre é fácil a tarefa do mapeamento do espaço de possíveis soluções para o problema, podendo haver a necessidade de envolver heurísticas adicionais como codificadores (FERREIRA, 2003).

2.1.2 População de Cromossomos

A população é constituída por um conjunto de cromossomos que são as possíveis soluções do problema. Normalmente a geração da população inicial é feita de maneira aleatória.

Um parâmetro de extrema importância para a execução do AG é o tamanho da população. Quando uma população é pequena, pode ocorrer uma estagnação no processo evolutivo, dependendo da evolução e da pouca variabilidade genética. Já uma população grande demais, poderá tornar o algoritmo extremamente lento e com baixo rendimento em termos de processamento computacional.

2.1.3 Fitness

A função de *fitness* (*f)* pode ser pensada como uma medida de desempenho, lucratividade, utilidade e excelência que se queira maximizar (GOLDBERG, 1989). O *fitness* é associado ao grau de resistência e adaptabilidade ao meio onde o indivíduo vive. Com isso, indivíduos com maior *fitness* terão maior chance de sobreviver e serão responsáveis pela próxima geração. Nos Algoritmos Genéticos é associado um valor numérico de adaptação, o qual se supõe que é proporcional à sua "utilidade" ou "habilidade" em função do seu objetivo.

2.1.4 Seleção

A seleção é feita após o calculo do *fitness*, no final são escolhidos alguns indivíduos para serem aplicados os operadores genéticos. A seleção considera o valor do *fitness*, ou seja, o indivíduo com maior *fitness* tem maior probabilidade de formar descendentes melhor adaptados.

Geralmente os Algoritmos Genéticos utilizam um método de seleção para obter um par de indivíduos que se tornarão os pais de dois novos indivíduos para a nova geração (FERREIRA, 2003).

A seleção pode ser feita utilizando o Método da Roleta, no qual cada indivíduo recebe um valor, que analogamente seria sua porção na roleta. Esse valor é uma proporção entre o

seu *fitness* e a soma dos *fitness* da população. Isso faz com que o indivíduo que tenha maior *fitness*, consequentemente tenha maior chance de ser escolhido (COSTA e BRUNA, 2002).

No método da roleta, como a probabilidade de ser escolhido não é 100%, há a possibilidade de que o melhor indivíduo não seja escolhido para a próxima geração. Para evitar isso, pode-se utilizar uma estratégia chamada de Estratégia Salvacionista (Elitismo), na qual uma porcentagem da população com os melhores *fitness* é preservada para a próxima geração automaticamente (COSTA e BRUNA, 2002).

Além do Método da Roleta, existem outros métodos que podem ser implementados para se efetuar a seleção, dentre os quais (COSTA e BRUNA, 2002):

- **Integral**: respeita rigidamente o *fitness* relativo.
- **Ranking**: indivíduos são ordenados de acordo com seu escore de *fitness*, e a chance de que cada indivíduo tem de ser selecionado é dada pela posição do ranking, não mais pelo valor absoluto do *fitness*. Este método pode resultar em convergência acelerada e manter a variância da probabilidade de seleção dos indivíduos de cada população a mesma.
- **Torneio**: indivíduos selecionados aleatoriamente disputam um torneio, no qual o melhor é selecionado para a reprodução. Este método é mais eficiente computacionalmente que o *ranking* e também diminui a probabilidade de convergência acelerada.
- **Elitismo:** força-se o algoritmo genético a reter o melhor indivíduo da geração atual, ou alguns dos melhores, na próxima geração.
- **Aleatória**: são selecionados aleatoriamente *N* indivíduos da população.

2.1.5 Operadores Genéticos

O principal objetivo dos operadores genéticos é, através de sucessivas gerações, transformar a população até chegar a um resultado satisfatório. Para que a população se diversifique e mantenha características de adaptação adquiridas pelas gerações anteriores é que os operadores são necessários.

Também são necessários para a introdução e manutenção da diversidade genética da população, alterando arbitrariamente um ou mais componentes de uma estrutura escolhida. Os operadores mais comuns são Cruzamento (*Crossover*) e Mutação, apresentados a seguir.

- **Cruzamento ou *Crossover***

Através da combinação de dois ou mais indivíduos, criam-se novos indivíduos. O cruzamento é feito a partir da troca de informação entre diferentes soluções candidatas. No algoritmo genético clássico é atribuída uma probabilidade de cruzamento fixa aos indivíduos da população.

O operador de cruzamento mais comumente empregado é o cruzamento de um ponto. Para a aplicação deste operador, são selecionados dois indivíduos (pais) e a partir de seus cromossomos são gerados dois novos indivíduos (filhos). Para gerar os filhos, seleciona-se um mesmo ponto de corte aleatoriamente nos cromossomos dos pais, e os segmentos de cromossomo criados a partir do ponto de corte são trocados. Considere dois indivíduos selecionados como pais a partir da população inicial de um algoritmo genético e suponha que o ponto de corte escolhido (aleatoriamente) encontra-se entre a posição 5 dos cromossomos dos pais (Figura 2.3):

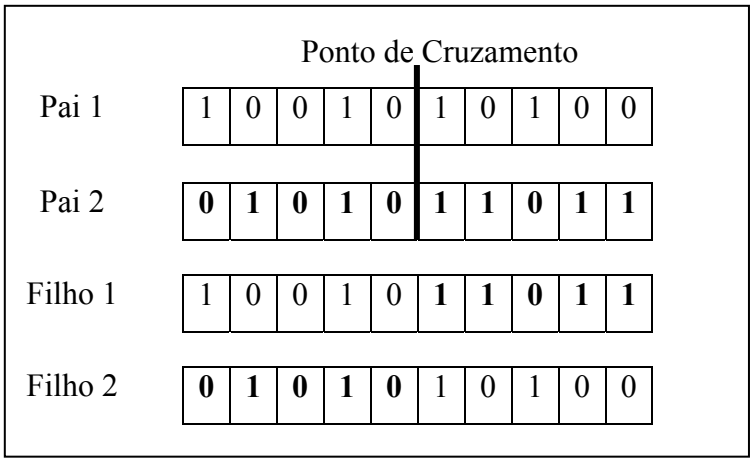


Figura 2.3: Ilustração do cruzamento de ponto único.

Há também a versão do cruzamento de dois pontos, na qual são escolhidas duas posições aleatoriamente do cromossomo e o segmento entre esses pontos é trocado, como pode ser visto na Figura 2.4.

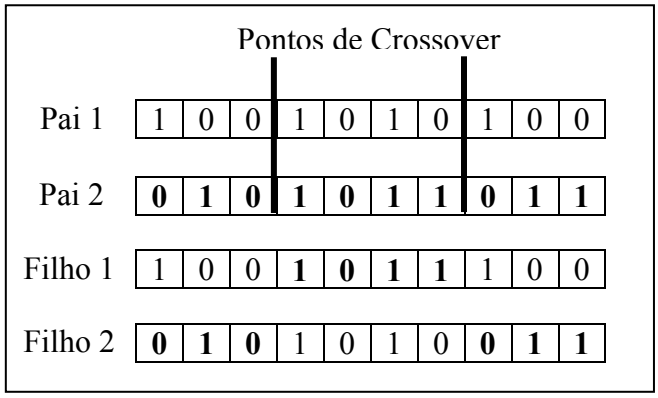


Figura 2.4: Ilustração de *cruzamento* de dois pontos.

- ***Mutação***

O operador de mutação modifica aleatoriamente um ou mais genes de um cromossomo. Usualmente, são atribuídos valores pequenos para a taxa de mutação. É um processo muito útil, pois cria uma diversidade e variabilidade extra na população, mas sem atrapalhar o progresso já alcançado no Algoritmo Genético.

2.2 Considerações Finais

Este capítulo apresentou os principais fundamentos dos Algoritmos Genéticos: como a população é representada, principais passos, operadores e estratégias que podem ser utilizados.

No capítulo a seguir são apresentados fundamentos do teste de software, outra área objeto desta pesquisa.

3 TESTE DE SOFTWARE

O processo de desenvolvimento de software envolve uma série de atividades nas quais, muito embora diferentes técnicas, métodos e ferramentas sejam utilizados, erros no produto ainda podem ocorrer (MALDONADO et al., 1998). Por isso o teste de software é uma atividade fundamental.

A atividade de teste é o processo de execução de um programa com a intenção de descobrir defeitos. Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar algum defeito ainda não descoberto. Caso no qual o teste é dito bem sucedido (MYERS, 1979).

O teste de software concentra-se em encontrar defeitos introduzidos durante qualquer fase do desenvolvimento ou manutenção de sistemas de software, podendo estes defeitos serem decorrentes de omissões, inconsistências ou mau entendimento dos requisitos ou especificações por parte do desenvolvedor (McGREGOR e SYKES, 2001). Entretanto, o teste não pode garantir que o software esteja livre de erros, mas o teste pode ser usado para aumentar a confiabilidade do software.

A IEEE tem realizado vários esforços de padronização, entre eles padronizar a terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE número 610.12-1990 diferencia os termos (MALDONADO et al., 1998):

- **Defeito** (*fault*) – passo, processo ou definição de dados incorretos, como por exemplo, uma instrução ou comando incorreto;
- **Engano** (*mistake*) – ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo programador;
- **Erro** (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e
- **Falha** (*failure*) – produção de uma saída incorreta com relação à especificação.

De acordo com McGregor e Sykes (2001), durante muito tempo, o teste de software era aplicado somente ao final do desenvolvimento dos sistemas. Devido aos altos custos associados à correção de erros encontrados e na manutenção do software, técnicas

sistemáticas de teste começaram a ser aplicadas em paralelo ao longo do processo de desenvolvimento.

A atividade de teste engloba a escolha dos dados de entrada (dados de teste), a execução dos dados escolhidos e, a comparação dos resultados obtidos com os resultados esperados para determinar se um defeito está presente ou não. Um caso de teste é composto pela entrada para o programa (dado de teste) mais a saída esperada.

Segundo Rapps e Weyuker (1985), um critério de teste é um predicado a ser satisfeito que estabelece certos requisitos de teste, chamados elementos requeridos, que devem ser exercitados durante a execução do programa, ou seja, executados por um dado de teste.

Um conjunto de teste é dito adequado com respeito a um dado critério se todos os casos de teste desse conjunto satisfizerem o critério, ou seja se cobrirem todos elementos requeridos pelo critério dado.

A atividade de teste pode ser considerada encerrada quando todos os elementos requeridos por um determinado critério de teste tiverem sido cobertos.

3.1 Fases do Teste de Software

O teste de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes. Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento de software (PRESSMAN, 1997). A atividade de teste pode ser realizada em quatro fases:

1. **Teste de Unidade:** concentra esforços na menor unidade do projeto de software e tem como objetivo garantir que os requisitos de implementação de cada unidade estejam corretos.
2. **Teste de Integração:** é uma atividade sistemática aplicada durante a integração da estrutura do programa visando a descobrir erros associados às interfaces entre os módulos; o objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto (MALDONADO et al., 1998).
3. **Teste de Validação:** verifica se os critérios de validação estabelecidos durante a especificação de requisitos estão corretos e, com isso garantir que o software

atenda a todas as exigências funcionais, comportamentais e de desempenho desejadas (VINCENZI et al., 2007).

4. **Teste de Sistema:** é realizado após a integração do sistema, visa a identificar erros de funções e características de desempenho que não estejam de acordo com a especificação.

Os critérios para teste de software aplicados em cada fase do teste estão baseados em três técnicas: Técnica Estrutural, Técnica Funcional e Técnica Baseada em Erros. Estas técnicas são descritas a seguir.

3.2 Teste Estrutural

O teste estrutural, também conhecido como teste caixa-branca ou *white-box testing*, é um tipo de teste no qual os casos de teste são identificados baseando-se na implementação do sistema. O objetivo do teste estrutural é testar detalhes procedimentais (MYERS, 1979). Por basear-se na implementação, o teste estrutural consegue testar partes do sistema que não estão na especificação, mas por outro lado, o teste estrutural não consegue identificar comportamentos que estão na especificação, mas não foram implementados. A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de programa (MALDONADO, 1991).

Segundo Rapps e Weyuker (1985), a representação de um programa *P* como um grafo de fluxo de controle consiste em um grafo orientado, com um único nó de entrada e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características:

- uma vez que o primeiro comando do bloco é executado, todos os demais são executados sequencialmente; e
- não existe desvio de execução para nenhum comando dentro do bloco.

Através do grafo de programa podem ser escolhidos os elementos que devem ser executados, caracterizando assim o teste estrutural. A Figura 3.1 contém o código fonte da função $\sum_{i=1}^{n} i^2$, que servirá de base para exemplificar os critérios estruturais.

```
void main ( )
  {
      f loat soma ;
      int i , n ;
      scanf ( "%d" , &n) ;                    /* nó 1 */
      soma = 1 ;                              /* nó 1 */
      for ( i = 0 ; i < n ; i++)              /* nós 1 , 2 e 3 */
          soma = soma + pow( i , 2) ;         /* nó 3 */
      printf ( "%f" , soma ) ;                /* nó 4 */
  }
```

Figura 3.1: Exemplo código fonte da função $\sum^{n}_{i=1} i^2$.

Pode-se destacar como exemplo de critérios estruturais:

- **Critérios Baseados em Fluxo de Controle**: utilizam somente características de controle da execução do programa, tais como comandos ou desvios, para derivar os requisitos de teste. A Figura 3.2 ilustra o grafo de fluxo de controle baseado no código da Figura 3.1. Os critérios mais conhecidos dessa classe são (PRESSMAN, 1997):
    - **Todos-Nós**: requer que a execução do programa passe pelo menos uma vez em cada nó do programa, fazendo com que todos os comandos do programa sejam executados;
    - **Todos-Arcos**: exige que a execução exercite cada arco do grafo de programa pelo menos uma vez, fazendo com que todos os desvios de fluxo de controle do programa sejam executados;
    - **Todos-Caminhos**: requer que todos os caminhos possíveis, derivados do grafo do programa sejam executados.

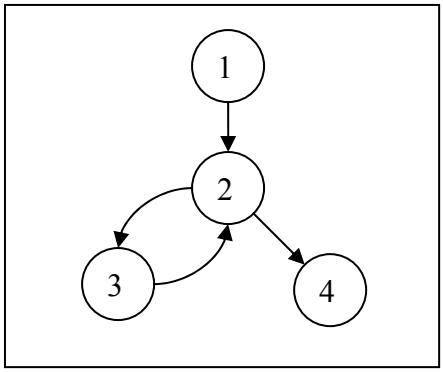


Figura 3.2: Grafo de fluxo de controle.

- **Critérios Baseados na Complexidade**: derivam os requisitos de teste a partir da complexidade do programa. O critério mais conhecido desta classe é o critério de McCabe, que requer a execução de um conjunto de caminhos linearmente independentes do grafo de programa, utilizando o conceito de complexidade ciclomática para derivar os requisitos de teste (PRESSMAN, 1997).

Para o entendimento dos critérios baseados em fluxo de dados, são necessários os seguintes conceitos relacionados com variáveis e caminhos (MALDONADO et al., 1998):

- **Definição de variável**: ocorre ao se armazenar um valor em uma posição de memória, e é representada no grafo de fluxo de dados (Figura 3.3) pelo nome da variável sucedido pelo símbolo ←.
- **Uso computacional (ou c-uso)**: é a referência ao valor de uma variável que afeta diretamente o resultado de uma computação, e é representado no grafo de fluxo de dados (Figura 3.3) pelo nome da variável sucedido pelo símbolo →.
- **Uso predicativo (ou p-uso)**: é a referência ao valor de uma variável que afeta diretamente o fluxo de controle, e é representado no grafo de fluxo de dados (Figura 3.3) pelo nome da variável adjacente a aresta correspondente ao desvio.

Um caminho da forma ($i, n_1, \ldots, n_m, j$) é dito livre de definição se há definição da variável x em i e não há definição de x nos nós $n_1 \ldots n_m$. Um caminho é dito simples se todos os nós, exceto possivelmente o primeiro e o último, são distintos. Um caminho é dito livre de laços se todos os seus nós são distintos (MALDONADO, 1991).

- **Critérios Baseados em Fluxo de Dados**: utiliza informações do fluxo de dados do programa para derivar os requisitos de teste. A Figura 3.3 ilustra como é um grafo de fluxo de dados, baseado no código da Figura 3.1. Os critérios mais conhecidos desta classe são:
    - **Todos-Usos**: requer que todas as associações entre uma definição de variável e seus usos (computacional ou predicado) subsequentes sejam exercitados por ao menos um caminho onde a variável não seja redefinida, caminho livre de definição (RAPPS e WEYUKER, 1985).
    - **Todos-Potenciais-Usos**: requer que todas as associações entre uma definição de variável e seus potencias usos (computacional ou predicado) subsequentes sejam exercitados por ao menos um caminho livre de definição (MALDONADO, 1991). Um potencial uso em um nó

j (ou arco (j, k)) é caracterizado pela existência de um caminho livre de definição até j ou (j, k) independentemente de existir um p-uso nesse nó (ou arco).

- **Todos-Potenciais-Usos/du**: requer que todas as associações entre uma definição de variável e seus potencias usos (computacional ou predicado) sejam exercitados ao menos por um caminho livre de definição e que seja livre de laço, chamado de du-caminho (MALDONADO, 1991).
- **Todos-Potenciais-du-Caminhos**: requer que todos os potenciais du-caminhos sejam exercitados (MALDONADO, 1991).

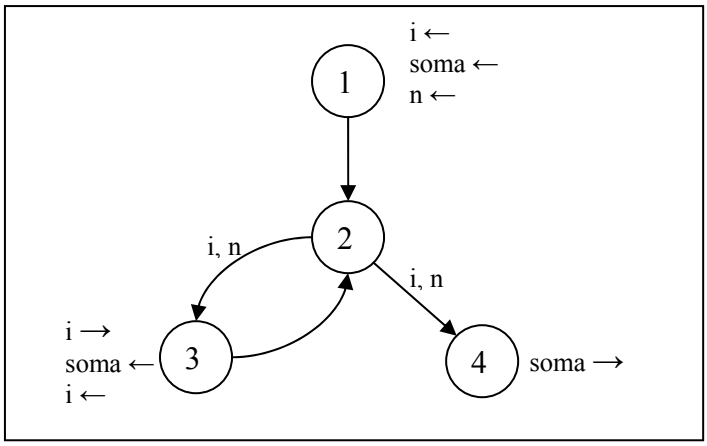


Figura 3.3: Grafo de fluxo de dados.

A Tabela 3.1 apresenta os elementos requeridos, para o exemplo da Figura 3.1, em relação aos critérios estruturais.

Tabela 3.1: Elementos requeridos para o exemplo da Figura 3.1.

Critério	**Elementos Requeridos**
Todos-Nós	1, 2, 3, 4
Todas-Arestas	(1,2) (2,3) (3,2) (2,4)
Todos-c-usos	(1, 4 {*soma*}), (1, 3, {*soma,i*}), (3, 3, {*soma, i*}), (3, 4, {*soma*})
Todos-p-usos	(1, (2, 3),{*n, i*}), (1, (2, 4),{*n, i*}), (3, (2, 3),{*i*}), (3,(2, 4),{*i*})
Todos-Usos	(1, 4, {*soma*}), (1, 3, {*soma, i*}), (3, 3, {*soma,i*}), (3,4,{soma}), (1, (2, 3), {*n, i*}), (1, (2, 4), {*n, i*}), (3, (2, 3), {*i*}), (3, (2, 4), {*i*})
Todos-du-caminhos	(1, 2, 4), (1, 2, 3) (3, 2, 4) (3, 2, 3)
Todos-Potenciais-usos	(1, 2, {*soma, i, n*}), (1, 3, {*soma, i, n*}), (1, 4, {*soma, i, n*}), (3, 2, {*soma, i*}), (3, 3, {*soma, i*}), (3, 4, {soma, i}), (1, (1, 2), {*soma, i, n*}), (1, (2, 3), {*soma, i, n*}), (1, (2, 4), {*soma, i, n*}), (1, (3, 2), {*n*}), (3, (2, 3), {*soma, i*}), (3, (2, 4), {*soma, i*}), (3, (2, 3), {*soma, i*})
Todos-Potenciais-Usos/Du	(1, (1, 2), *n*), (1, (1, 2), *soma*), (1, (1, 2), *i*), (1, (2, 3), *n*), (1, (2, 3), *i*), (1, (2, 4), *n*), (1, (2, 4), *i*), (1, (2, 4), *soma*), (3, (2, 3), *n*), (3, (2, 3), *i*), (3, (3, 2), *i*), (1, (2, 4), soma) (3, (2, 4), *n*), (3, (2, 4), *i*), (3, (2, 4), *soma*)
Todos-Potenciais-Du-Caminhos	(1, 2, 3) (1, 2, 4) (1, 2, 3, 2, 4) (1, 2)

3.3 Teste Funcional

O teste funcional, também conhecido como caixa-preta ou *black-box testing* (MYERS, 1979) é realizado baseando-se apenas nos requisitos do software. É um tipo de teste realizado com o objetivo de observar se, para uma dada entrada, o software produz a saída correta.

Os dois principais passos desse teste são: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software (PRESSMAN, 1997).

Uma importante vantagem é que as atividades de teste podem ser executadas em paralelo ao desenvolvimento da aplicação, contribuindo para um melhor entendimento e correção dos modelos e especificações desde as etapas iniciais dos processos, evitando

detecções de problemas tardiamente, diminuindo assim, o impacto e custos associados às mudanças (JORGENSEN, 2001).

Exemplos de critérios de teste funcional são (PRESSMAN, 1997):

- **Particionamento de Equivalência:** divide o domínio de entrada em classes de dados válidas e inválidas que provavelmente exercitarão uma função de software específica e exige a execução de pelo menos um dado de teste em cada classe.
- **Análise do Valor Limite:** verifica a capacidade de um programa em manipular dados nos limites das condições de entrada, exige a execução de dados de teste dos limites de cada classe de equivalência.
- **Grafo de Causa-Efeito**: oferece uma representação concisa das condições lógicas e das ações correspondentes possibilitando que o testador valide conjuntos de ações e condições.

3.4 Teste Baseado em Defeitos

Esta técnica utiliza informações sobre os erros mais frequentes cometidos no processo de desenvolvimento do software, com isso pode-se verificar a ausência ou presença de erros nos programas. Os critérios mais conhecidos são:

- **Semeadura de Erros**: coloca-se uma determinada quantidade de erros no programa, após é efetuado o teste, do total de erros encontrados, verificam-se quais são novos (naturais) e quais são antigos (artificiais, colocados anteriormente). O número de erros naturais pode ser estimado utilizando estimativas de probabilidade (GOEL, 1985).
- **Análise de Mutantes**: é baseado em dois pressupostos: i) na hipótese do programador competente, que diz que os programadores desenvolvem programas próximos do correto e, ii) no efeito do acoplamento, ou seja, defeitos complexos são revelados revelando-se defeitos simples (DeMILLO et al., 1978). A idéia deste critério é inserir pequenas modificações no programa em teste através dos chamados operadores de mutação. Cada programa modificado é denominado mutante. Dados de teste são executados com o propósito de que todos os mutantes criados sejam mortos, ou seja, deve-se executar o programa mutante com um dado de teste de tal forma que sua saída seja diferente da saída do programa original

para aquele dado. Os que não forem mortos são considerados vivos e caso não exista um caso de teste que diferencie o comportamento do programa mutante do comportamento do programa original este é dito mutante equivalente.

3.5 Ferramentas

Com o uso de ferramentas é possível reduzir o esforço necessário para a realização do teste, e também diminuir os erros que são causados pela intervenção humana nessa atividade. Além disso, a existência de ferramentas de teste viabiliza a realização de estudos empíricos com o objetivo de avaliar o custo e a eficácia das técnicas e critérios de teste.

Em geral uma ferramenta de teste gera os elementos requeridos por um critério, os quais o usuário tentará cobrir com a entrada de dados de teste. O usuário adiciona dados de teste para aumentar a cobertura dos elementos requeridos para os critérios desejados.

Entre as ferramentas que automatizam o teste de software destacam-se: POKE-TOOL (CHAIN, 1991) e PROTEUM (DELAMARO e MALDONADO, 1996).

3.5.1 Ferramenta POKE-TOOL

A ferramenta POKE-TOOL (*POtential Uses CRIteria TOOL for program testing*) (CHAIN, 1991), apóia a aplicação dos critérios estruturais, baseados em fluxo de controle e baseados em fluxo de dados, para funções escritas na linguagem C.

A POKE-TOOL é compreendida por três programas: *poketool, pokeexec* e *pokeaval*. O programa *poketool* gera a lista de elementos requeridos e o programa instrumentado. Depois de compilado, o programa instrumentado é executado pelo programa *pokeexec*, que armazena a informação referente aos elementos cobertos pelos dados de teste. O programa *pokeaval*, faz a checagem dos elementos requeridos por um critério de seleção escolhido pelo usuário com os elementos cobertos pelos casos de teste, informando ao usuário quais elementos ainda não foram cobertos. Isto possibilita ao usuário realizar novos testes a fim de maximizar a cobertura para o critério escolhido.

3.5.2 Ferramenta PROTEUM

A ferramenta PROTEUM (*PROgram Testing Using Mutantes*) (DELAMARO e MALDONADO, 1996) foi desenvolvida no ICMC-USP para apoiar o critério de Análise de Mutantes (De MILLO et al., 1978), é uma ferramenta multi-linguagem sendo mais comumente utilizada em programas escritos na linguagem C.

As principais funções da ferramenta PROTEUM correspondem às atividades necessárias para a aplicação do critério Análise de Mutantes: definição de casos de teste; simulação da especificação em teste; geração de mutantes; execução dos mutantes; análise dos mutantes vivos e cálculo do escore de mutação.

A ferramenta PROTEUM conta com 71 operadores para o teste de unidade. O conjunto de operadores de mutação de unidade é classificado em quatro classes: Mutação de Constantes, Mutação de Comandos, Mutação de Operadores e Mutação de Variáveis (DELAMARO e MALDONADO, 1996).

Após a execução a ferramenta mostra o status do teste com informações como: número total de mutantes, número de mutantes vivos, escore de mutação, entre outras. Com estes dados, o testador pode inserir novos casos de teste para melhorar a qualidade dos dados de teste, podendo este processo ser repetido até que um escore desejado seja alcançado.

3.6 Geração de Dados de Teste[1]

É impraticável testar todos os possíveis valores de entrada ou exercitar todos os caminhos de um programa. A tarefa de geração de dados de teste visa a escolher pontos do domínio de entrada para satisfazer um dado critério. A geração de dados de teste é considerada um problema indecidível, sendo que existem muitas restrições inerentes às atividades de teste que impossibilitam automatizar completamente essa etapa. Destacam-se:

- **Correção Coincidente**: ocorre quando o programa em teste possui um defeito que é alcançado por um dado de teste, um estado de erro é produzido, mas coincidentemente um resultado correto é obtido.
- **Caminho Ausente**: corresponde a uma determinada funcionalidade requerida para o programa, mas que por engano não foi implementada, isto é, o caminho correspondente não existe no programa.

[1] Esta parte foi parcialmente extraída de (VERGILIO et al., 2007).

- **Caminhos Não Executáveis**: um caminho é dito não executável se não existe um conjunto de valores atribuídos às variáveis de entrada do programa, parâmetros e variáveis globais que causam a sua execução. Um elemento requerido por um dado critério estrutural é não executável se não existe caminho executável que o cobre.
- **Mutantes Equivalentes:** o programa mutante implementa a mesma função que o programa original e portanto não se pode gerar um dado de teste para diferenciá-los. Entretanto, detectar a equivalência destes programas também é, em geral, indecidível.

Apesar das limitações existentes, encontram-se na literatura diferentes categorias de técnicas que podem ser utilizadas para gerar dados de teste para satisfazer os diversos critérios. As mais conhecidas são:

- geração aleatória de dados – pontos do domínio de entrada são selecionados aleatoriamente até que o critério de teste seja satisfeito (DURAN e NTAFOS, 1984);
- geração com execução simbólica – tem o objetivo de auxiliar a geração automática de dados de teste para um dado caminho ou conjunto de caminhos (CLARKE, 1976; HOWDEN, 1987; BOYER et al., 1975; RAMAMOORTHY et al., 1976);
- geração com execução dinâmica – está baseada na execução real do programa em teste, em métodos de minimização de funções e análise de fluxo de dados (KOREL, 1990);
- geração de dados sensíveis a defeitos – fundamentos permitem escolher pontos do domínio de entrada relacionados a certos tipos de defeitos e, por isso, com alta probabilidade de que esses sejam revelados, são exemplos dessa técnica, Teste Domínios (WHITE e COHEN, 1980), Teste Baseado em Restrições (DeMILLO e OFFUTT, 1991) .
- geração utilizando técnicas de computação evolutiva que é o foco desse trabalho e por isso será descrita a seguir.

3.6.1 Geração com Algoritmos Evolutivos

Alguns autores propuseram a utilização de técnicas de computação evolutiva (Capítulo 2) para lidar com as limitações da atividade de geração de dados de teste, dando origem a uma nova área de pesquisa chamada Teste Evolucionário (WEGENER et al., 2001).

Na literatura muitos trabalhos utilizam Algoritmos Genéticos para a geração de dados de teste. A maioria deles visa à geração de dados de teste para critérios baseados em fluxo de controle (McGRAW e MICHAEL, 1997; MICHAEL et al., 2001; JONES et al., 1996; PARGAS et al., 1999; ROPER et al., 1995; TRACEY et al., 1998; WEICHSELBAUM, 1998). Alguns visam a satisfazer critérios baseados em fluxo de dados (BUENO e JINO, 2001; FERREIRA e VERGILIO, 2005; WEGENER et al., 2001). Outros focalizam o critério análise de mutantes (BOTTACI, 2001; CHUNG, 1998; FERREIRA e VERGILIO, 2005). Esses trabalhos podem ser classificados em duas categorias dependendo da função de avaliação utilizada:

1. **Função de Avaliação Baseada na Cobertura:** os indivíduos na população codificam valores possíveis para as variáveis de entrada do programa em teste, e a função de *fitness* de cada indivíduo está associada à cobertura de um dado critério de teste.
2. **Função de Avaliação Orientada a um Objetivo:** geralmente considera a execução dinâmica de programas e um objetivo que orienta a função de avaliação, que é o de executar um caminho particular do programa para cobrir um dado elemento requerido por um critério de teste em particular.

3.7 Considerações Finais

Esse capítulo apresentou os principais conceitos associados ao teste de software, objetivos, etapas, critérios e ferramentas. Uma revisão de trabalhos que utilizam técnicas de computação evolutiva para gerar dados de teste também foi apresentada.

Todos os conceitos e idéias apresentadas foram adaptados ao contexto de software orientado a objetos, no qual novos critérios e ferramentas foram propostos e serão descritos no próximo capítulo.

O teste evolutivo tem sido trabalhado, como mencionado, por muitos trabalhos e diversas técnicas de computação evolutiva foram exploradas. Os resultados mostram que

essas aplicações são bem sucedidas no teste de programas procedurais. Entretanto, no contexto de orientação a objetos poucos trabalhos existem. Estes trabalhos também são assunto do próximo capítulo que trata especificamente de software orientado a objetos.

4 TESTE DE SOFTWARE ORIENTADO A OBJETOS

O software Orientado a Objetos (OO) é composto de classes e de objetos possuindo uma arquitetura baseada em várias camadas de subsistemas que encapsulam classes de colaboração, sendo que cada elemento do subsistema executa funções que auxiliam na cobertura dos requisitos do sistema. O teste de software orientado a objetos é realizado em diferentes níveis para descobrir defeitos à medida em que as classes colaboram umas com as outras (PRESSMAN, 2006).

O teste de software orientado a objetos é similar ao teste de software convencional, entretanto, deve enfocar sequências apropriadas de operações para exercitar os estados de uma classe (PRESSMAN, 2006). Para o teste orientado a objetos são necessárias abordagens especiais para lidar com herança, polimorfismo e acoplamento dinâmico.

Este capítulo trata especificamente do teste de software orientado a objetos descrevendo fases do teste, e os principais critérios de teste considerados nesse trabalho, assim como uma descrição dos principais trabalhos relacionados.

4.1 Fases de Teste OO

No capítulo anterior foram introduzidas as principais fases do teste. Quando consideramos o teste de software orientado a objetos mais ênfase deve ser dada ao teste de integração devido ao grande número de métodos e de conexões existentes, pois o software orientado a objetos é composto de classes que encapsulam uma série de métodos, geralmente de baixa complexidade, que cooperam entre si na implementação de uma dada funcionalidade (JORGENSEM e ERICKSON, 1994; BINDER, 1999).

Segundo Colanzi (1999), o teste orientado a objetos é organizado em quatro fases:

1. Teste de unidade: testa os métodos individualmente;
2. Teste de classe: testa a interação entre métodos de uma classe;
3. Teste de integração: testa a integração entre classes do sistema;
4. Teste de sistema: testa a funcionalidade do sistema como um todo.

Na definição acima, o método é considerado a menor unidade a ser testada. Entretanto, no contexto de orientação a objetos, a classe que constitui uma aplicação também é considerada por muitos autores como a menor unidade que pode ser testada isoladamente.

A Tabela 4.1, (extraída de VINCENZI et al., 2007) sintetiza os tipos de teste que podem ser aplicados em cada uma das fases tanto nos programas procedimentais quanto em programas OO considerando o método ou a classe como sendo a menor unidade.

Tabela 4.1: Relação entre fases de teste de programas procedimentais e Orientados a Objetos (VINCENZI et al., 2007).

Menor Unidade: Método		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método
Integração	Interprocedimental	Inter-método, Intra-classe e Inter-classe
Sistema	Toda Aplicação	Toda Aplicação
Menor Unidade: Classe		
Fase	Teste Procedimental	Teste Orientado a Objetos
Unidade	Intraprocedimental	Intra-método, Inter-classe e Intra-classe
Integração	Interprocedimental	Inter-classe
Sistema	Toda Aplicação	Toda Aplicação

No teste OO, conjuntos de critérios de teste, como cobertura de ramos, são usados como resposta para os cenários de teste e determinam quando o processo de geração de casos de teste pode terminar. A definição desses critérios foi feita, originalmente, para o teste de programas procedimentais, mas vêm sendo estendidas ao longo dos anos para se adequarem ao teste de programas OO.

Segundo Wappler e Wegener (2006), o teste de uma classe frequentemente envolve outras classes, por exemplo, classes que aparecem como parâmetro em métodos de classes sendo testados (CUT – *Class Under Test*). O conjunto de classes que é relevante para testar uma classe em particular é chamado *cluster*.

O teste de unidade para software orientado a objetos consiste na sequência de chamadas de métodos e um ou mais comandos. A sequência de métodos representa o cenário de teste. Durante a execução, todos os objetos participantes do teste são criados e colocados em um estado particular por várias chamadas de métodos para estes objetos. Os comandos verificam se o sistema está em um estado válido depois da execução da sequência de

chamadas de métodos. Quando se testa uma classe orientada a objetos usando um critério de cobertura adequado, casos de teste devem ser gerados para que satisfaçam o critério para todos métodos dessa classe. Assim, cada caso de teste foca na execução de um método em particular, o método sendo testado (MUT – *Method Under Test*). Consequentemente, todos casos de teste adequados para cada método da classe que está sendo testado satisfazem adequadamente o critério para toda a classe (WAPPLER e WEGENER, 2006).

O sistema todo é integrado, depois da realização dos testes inter e intra classe, podendo ser realizados os teste de sistema que, por serem baseados em critérios funcionais, não apresentam diferença fundamentais entre o teste procedimental e Orientado a Objetos (VINCENZI et al., 2007).

4.2 Critérios de Teste OO

Muitos são os trabalhos que introduzem critérios de teste considerando o contexto de orientação a objetos, os critérios podem estar baseados na especificação ou no programa. Os primeiros (BINDER, 1999; CHEN et al., 1998; HOFFMAN e STOOPER, 1997; OFFUT e IRVINE, 1995) consideram diversos diagramas do projeto orientado a objetos para derivar os requisitos de teste tais como os diagramas de estado que modelam comportamento do objeto, diagramas de caso de uso e etc.

Dentre os trabalhos que realizam o teste baseado em programas são de especial interesse os que estendem os critérios estruturais para o contexto de software OO. Dentre estes destaca-se o trabalho de Harold e Rothermel (1994), que considera defeitos tipos de interações de fluxo de dados em uma classe. Os autores comentam que os critérios de fluxo de dados destinados ao teste de programas procedimentais podem ser utilizados tanto para o teste de métodos individuais quanto para o teste de métodos que interagem entre si dentre de uma mesma classe. Entretanto, esses critérios não consideram interações de fluxo de dados quando os usuários de uma mesma classe invocam sequência de métodos em uma ordem arbitrária.

Para viabilizar o teste de fluxo de dados nos níveis intra-método, inter-método e intra classe, Harrold e Rothermel (1994), propuseram as seguintes representações de programas: grafo de chamadas de classe (*class call graph*), grafo de fluxo de controle de classe (*CCFG – Class Control Flow Graph*) e o *framed* CCFG. Com base nessas representações, os quatro níveis de teste foram considerados (HARROLD e ROTHERMEL, 1994):

- **Teste intra-método**: testa os métodos individualmente. Esse nível é equivalente ao teste de unidade de programa procedimentais.
- **Teste inter-método**: testa os métodos públicos em conjunto com outros métodos dentro de uma mesma classe. Esse nível de teste é equivalente ao teste de integração de programas procedimentais.
- **Teste intra-classe**: testa interações entre métodos públicos fazendo chamada a esses métodos em diferentes sequências. O objetivo é identificar possíveis sequências de ativação de métodos inválidas que levem o objeto a um estado inconsistente.
- **Teste inter-classe:** usa o conceito de invocação de métodos públicos em diferentes sequências, entretanto, esses métodos públicos não necessitam estar na mesma classe.

O teste de mutação também foi proposto inicialmente para o teste de unidade de programas procedimentais. Entretanto, já existem extensões desse critério tanto para o teste de integração quanto para o teste de especificações. A flexibilidade do critério vem do fato de que para se aplicar o teste de mutação é necessária a existência de um modelo que seja executável e que aceite uma entrada e produza uma saída que possa ser comparada com a saída mutante (VINCENZI et al., 2007). Um exemplo de ferramenta que implementa o teste de mutação em um programa Java é a MuJava (MA et al., 2004).

4.3 Ferramenta JaBUTi (Java Bytecode Understanding and Testing)

A ferramenta JaBUTi (*Java Bytecode Understanding and Testing*), que apóia o teste estrutural em programas Java, foi desenvolvida por Vincenzi et al. (2003). Esta ferramenta funciona tanto com interface gráfica quanto na linha de comando. Com o uso desta ferramenta, que efetua toda a análise estática e dinâmica do programa em teste a partir do *bytecode* Java, é permitido ao usuário do componente avaliar a cobertura do seu conjunto de teste sobre o código do componente, sem que o código-fonte lhe esteja disponível.

JaBUTi automatiza parte das etapas necessárias para a aplicação da técnica de teste estrutural em código Java. Com a ferramenta é possível compreender e testar programas em código Java, apoiando a aplicação dos seguintes critérios: todos-nós, todos-ramos, todos-usos

e todos-potenciais-usos. Dentre as funcionalidades desta ferramenta, podem-se citar a criação dos grafos de fluxo de dados, apresentação dos elementos requeridos e análise de cobertura a partir do código objeto Java (VINCENZI et al., 2003).

O primeiro passo na utilização da JaBUTi é a criação de uma hierarquia de abstração para o programa que está sendo testado. Sabe-se que um programa Java é composto por um conjunto de classes organizadas em uma árvore hierárquica, tendo como classe raiz ***java.lang.Object***. O modelo utilizado pela JaBUTi delimita o programa a restringir um conjunto de classes. A partir de uma "classe base"- a que contém o ponto de entrada do programa - a ferramenta calcula um conjunto de classes necessárias excluindo: 1) Java API; 2) classes que não foram encontradas na classe caminho; e 3) pacotes que o testador decide ignorar (VINCENZI et al., 2003).

JaBUTi é usada na criação de um projeto de teste. O testador define um nome para o arquivo de projeto para armazenar os dados do mesmo.

Depois que o projeto de teste é criado, o testador tem oito critérios estruturais para trabalhar. Estes critérios são resumidos na Tabela 4.2.

Ao selecionar um critério, o testador visualiza informações relativas ao critério selecionado. Por exemplo, usando o critério todos-nós$_{ei}$ pode-se ver o código fonte (se disponível), o *bytecode* ou o grafo de fluxo de dados com as respectivas definções e usos de varíaveis. Em ambos os casos, são fornecidas dicas ao testador sobre quais requisitos devem ser cobertos para alcançar uma maior cobertura.

Tabela 4.2: Critérios de teste implementados pela JaBUTi (VINCENZI et al., 2007).

Nome	Significado	Explicação
Todos-nós$_{ei}$	Todos-nos independente de exceção.	Requer a execução de cada nó do grafo que pode ser executado sem a ocorrência de uma exceção.
Todos-ramos$_{ei}$	Todos-ramos independente de exceção.	Requer a execução de cada ramo do grafo que pode ser executado sem a ocorrência de uma exceção.
Todos-usos$_{ei}$	Todos-usos independente de exceção.	Requer a execução de cada par de def-uso que possa ser executado sem a ocorrência de uma exceção.
Todos-pot-usos$_{ei}$	Todos-pot-usos independente de exceção.	Requer a cobertura de todos os pares def-potencial-uso sem a ocorrência de uma exceção.
Todos-nós$_{ed}$ Todos-ramos$_{ed}$ Todos-usos$_{ed}$ Todos-pot-usos$_{ed}$	Mesmos critérios apresentados acima, mas dependente de exceção.	Requer, respectivamente, a cobertura de nós, ramos, pares def-uso e, a cobertura nós, bordas, pares def-uso e pares def-potencial-uso que só podem ser executados com a ocorrência de uma exceção

A ferramenta JaBUTi tem uma grande variedade de relatórios que permitem ao testador verificar os requisitos de cobertura de teste. Por exemplo, o testador pode criar um resumo da cobertura por critério, por classe ou método. Ele também pode criar arquivos HTML com dados estáticos ou dinâmicos obtidos até o momento no projeto.

Neste trabalho a versão script da ferramenta JaBUTi foi utilizada. Exemplos de relatórios gerados pela ferramenta e os principais comandos da versão script utilizados neste trabalho são apresentados respectivamente nos Apêndices A e B.

4.4 Trabalhos Relacionados

As técnicas de geração de dados de teste descritas no Capítulo 3 também podem ser utilizadas no contexto de teste de software OO. Entretanto, existem poucos trabalhos que consideram esse tema, ou seja, dado um conjunto de elementos requeridos por um critério específico para o contexto de programas OO, gerar dados de teste automaticamente para satisfazer o critério dado.

A completa automatização dessa etapa não é possível, pois neste contexto encontramos as mesmas limitações do teste em programas procedimentais tais como: caminhos não executáveis e mutantes equivalentes.

Nesta seção são descritos os principais trabalhos relacionados a geração de dados de teste para satisfazer critérios de teste no contexto de POO, e que utilizam algoritmos evolutivos e estão mais relacionados ao trabalho sendo proposto.

Como mencionado no Capítulo 3, Algoritmos Genéticos são aplicados com sucesso para geração de casos de teste para programas procedurais. Enquanto um caso de teste para programas procedurais consiste numa sequência de valores de entrada, a serem passados para o procedimento em execução, um caso de teste para um método de uma classe inclui a criação de um objeto, a mudança de estado interno, e finalmente inclui a invocação do método a ser testado, com valores de entrada apropriados (TONELLA, 2004). Isto é, quando se considera o teste de unidade de classes, o caso de teste deve levar em consideração a criação de um ou mais objetos, e a mudança de seus estados internos antes da invocação final

do método que está sendo testado. Muitos desses trabalhos conservam o formato de uma classe de teste do *framework* JUnit[2].

Um dos primeiros trabalhos sobre o teste evolucionário de software orientado a objetos foi o de Tonella (2004). Este trabalho representa as potenciais soluções através de cromossomos. Tonella propôs uma gramática, definiu um algoritmo genético com operadores genéticos especiais para recombinação e mutação. A população de indivíduos, representando os casos de teste, foi desenvolvida para aumentar uma medida da aptidão (*fitness*), levando em consideração a capacidade dos casos de teste satisfazer um critério de cobertura escolhido (todos-arcos). Novos casos de teste são gerados enquanto há objetivos a serem cobertos ou o tempo de execução máximo não é alcançado. O algoritmo proposto por Tonella é apresentado na Figura 4.1.

```
Gerar_CasosdeTeste(Classe_testada: Class)
1.  alvos_a_cobrir ← alvos(Classe_testada)
2.  População_atual ← gerar_população_aleatoriamente(tamanho_população)
3.  enquanto alvos_a_cobrir ≠ 0 e Tempo_execução < Tempo_máx_execução
4.      t ← alvo_selecionado(alvos_a_cobrir), tentativas ← 0
5.      enquanto (não coberto(t) e tentativas < Max_tentativas
6.          executar casos de teste na População_atual
7.          atualizar alvos_a_cobrir
8.          se cobriu(t) break
9.          computar fitness(t) para os casos de teste em População_atual
10.         extrair Nova_População da População_atual de acordo com fitness(t)
11.         aplicar operadores de mutação na Nova_População
12.         População_atual ← Nova_População
13.         tentativas ← tentativas + 1
14.     fim enquanto
15. fim enquanto
```

Figura 4.1: Algoritmo Genético extraído (TONELLA, 2004).

Segue a gramática ilustrada na Figura 4.2.

[2] JUnit: é um framework que auxilia a criação e execução de testes unitários sobre classes Java. Esse framework facilita a criação de código para a automação de testes com apresentação dos resultados.

```
<cromossomo>  ::=   <ações> @ <valores>
<ações>       ::=   <ação> {: <ações>} ?
<ação>        ::=   $id = construtor( {<parametros>}? )
                |   $id = classe # nula
                |   $id . metodo( {<parametros>}? )
<parametros>  ::=   <parametro> {, <parametros>? }
<parametro>   ::=   tipo_interno {<gerador>} ?
                |   $id
<gerador>     ::=   [maiuscula ; minuscula]
                |   [Classe]
<valores>     ::=   <valor> {, <valor>} ?
<valor>       ::=   inteiro
              ::=   real
              ::=   logico
              ::=   cadeia
```

Figura 4.2: Sintaxe do cromossomo (adaptada de (TONELLA, 2004)).

Um Algoritmo de Otimização de Colônia de Formigas foi proposto por Liu et al. (2005), no qual o foco estava na otimização de geração de sequências de chamadas mais curtas para um determinado objetivo de teste, sem violar as dependências de comportamento e encapsulamento.

Já Wappler e Lammermann (2005), utilizaram algoritmos evolutivos para o teste de unidade, focando o uso de algoritmos evolutivos universais[3]. A codificação proposta foi definida para representar testes de programas OO de modo que, tipos básicos de estrutura permitissem a aplicação de várias técnicas de busca como *Hill Climbing* ou *Simulated Annelaing.* A codificação sugerida, entretanto, não previa a geração de indivíduos que poderiam causar erros ao serem decodificados, a função de *fitness* usava diferentes mecanismos de penalidade a fim de penalizar sequências inválidas e guiar a busca para regiões do espaço da solução que continham sequências válidas. Para trabalhos futuros os autores indicam uma necessidade de investigação para o problema da função de controle de

[3] Algoritmos Evolutivos Universais: são algoritmos evolutivos fornecidos por caixas de ferramentas populares que são independentes do domínio de aplicação e oferecem várias pré-definições e operadores evolutivos probabilísticos (RIBEIRO et al., 2007).

distância dos nós[4] para operadores específicos no contexto de orientação a objetos. Além disso, esses trabalhos deveriam considerar uma maneira de cobrir métodos que não são públicos.

Por isso, em um trabalho posterior, Wappler e Lammermann (2006) lidaram com a questão da alcançabilidade de sequências. Eles utilizaram programação genética fortemente tipada, pois esta é capaz de expressar as dependências de chamadas de métodos para um dado objeto e a viabilidade das seqüências de chamada de método é preservado durante todo o processo de busca. O trabalho dos autores utilizou apenas quatro objetos de teste.

Seesing e Gross (2006) utilizaram a Programação Genética (PG) para geração automática de dados de teste para software orientado a objetos. A principal melhoria, ou vantagem, do método proposto sobre os outros algoritmos, é que o teste de software já está representado como um programa funcional. Um problema dessa abordagem é que a técnica apresentada não foi aplicada a problemas extensos, de modo que os experimentos realizados só podem ser considerado como uma prova de conceito.

Cheon et al. (2005) utilizaram a técnica funcional (caixa preta) para investigar o uso de Algoritmos Genéticos para geração de dados de teste, e as especificações de programa escritas em *Java Modeling Language* (JML é uma linguagem de especificação formal do comportamento de interface Java) foram usadas para determinar o resultado de teste. O compilador JML foi estendido para fazer o Java *bytecode* produzir informação sobre cobertura de teste.

Alguns experimentos foram realizados por Liaskos et al. (2007) alcançando excelentes resultados em critérios baseados em fluxo de dados.

Nos trabalhos citados, a representação de programas é baseada no código fonte; além disso, a instrumentação para extrair informação sobre a execução também é feita no nível de código fonte (RIBEIRO et al., 2007).

Ribeiro et al. (2007) utilizam uma representação para população que é baseada em *bytecode*. Em Ribeiro (2008) uma ferramenta denominada eCrash para geração de dados de teste é implementada. O indivíduo é codificado utilizando PG fortemente tipada. No final, cada árvore representa um grupo de funções.

[4] Função de distância dos nós: exprime o quão perto a execução do teste se aproxima do nó meta no grafo de fluxo de controle (WAPPLER e LAMMERMANN, 2005).

4.5 Considerações Finais

Esse capítulo apresentou os principais conceitos associados ao teste de software OO, fases, critérios e uma revisão dos principais trabalhos.

Os trabalhos mencionados neste capítulo, que possuem o objetivo de geração de dados de teste, utilizam diferentes técnicas de Aprendizado de Máquina e com diferentes objetivos. Alguns destes trabalhos não possuem o objetivo de satisfazer um critério de teste. Os que possuem tal objetivo abordam apenas critérios baseados em fluxo de controle, quase sempre o critério todos-nós ou todos-ramos. Além disso, exceto Ribeiro (2008), não oferecem uma implementação do algoritmo integrada com uma ferramenta de suporte ao critério considerado. Outra questão é que os algoritmos não consideram alguns aspectos específicos da atividade de teste em seus mecanismos de evolução.

No próximo capítulo é descrita uma ferramenta de geração de dados de teste baseada em AGs que satisfazem critérios para o *bytecode* e consideram aspectos específicos da atividade de teste.

5 FERRAMENTA IMPLEMENTADA

O objetivo deste trabalho é implementar uma ferramenta para gerar dados de teste para programas orientados a objetos e satisfazer os critérios baseados em *bytecode* implementados pela ferramenta JaBUTi. Permitindo a aplicação de uma estratégia que englobe diferentes critérios de teste de uma maneira complementar. Além disso, o apoio à geração automática permitirá a redução de custos e esforços gastos na atividade de teste.

A ferramenta implementada está baseada em AG e utiliza a representação proposta por Tonella (2004) e implementa um AG e um AG chamado de AG Ineditismo (AGI) com modificações incrementais para melhor atender à geração de um conjunto de dados de teste para os critérios citados acima. Essas modificações consideram características específicas da atividade de teste.

Este capítulo descreve a ferramenta implementada, chamada *Test Data Set Generator for OO* software *(*TDSGen/OO*),* para gerar dados de teste para satisfazer os critérios da ferramenta JaBUTi.

A Figura 5.1 apresenta a estrutura da TDSGen/OO. Esta estrutura é baseada em trabalho anterior (FERREIRA, 2003) e contém 4 módulos principais: Geração da População, Avaliação, Evolução e Início. Toda informação produzida pelos módulos são armazenados em arquivos, representados pelas elipses. Tudo começa pela leitura do arquivo de configuração que contém os parâmetros necessários para a geração da população inicial, após isso é feita a avaliação da cobertura dos dados de teste utilizando a ferramenta JaBUTi. Com esta avaliação é possível realizar a evolução, aplicando operadores genéticos e gerando uma nova população. Esse ciclo continua até que seja atingida a quantidade de gerações passada no arquivo de configuração. A descrição mais detalhada dos módulos são descritos nas próximas seções.

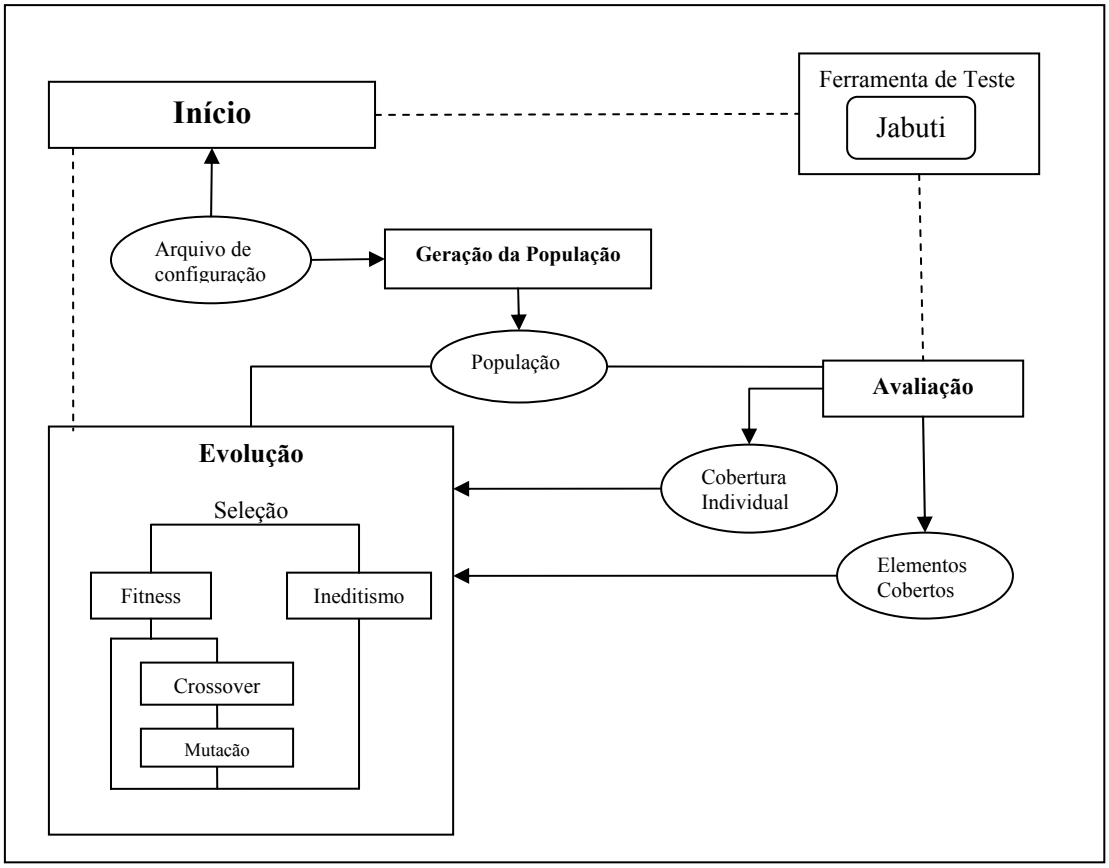


Figura 5.1: Estrutura de integração da ferramenta.

5.1 Geração da População

Este módulo é responsável pela geração de cada indivíduo que formará a população. Esta população inicial é gerada aleatoriamente de acordo com as especificações passadas no arquivo de configuração.

Um caso de teste é uma sequência de construtores e invocações de métodos, incluindo parâmetros. A representação escolhida para o indivíduo (cromossomo) utiliza a gramática introduzida por Tonella (2004) reapresentada aqui na Figura 5.2.

O cromossomo é dividido em duas partes, separado pelo caracter '@'. A primeira parte, não-terminal <ações> contém a sequência de construtores e invocação de métodos, separados pelo caracter ':' . A segunda parte contém os valores de entrada, separados pelo caracter ',' usados para cada invocação.

```
<cromossomo>  ::=   <ações> @ <valores>
<ações>       ::=   <ação> {: <ações>} ?
<ação>        ::=   $id = construtor( {<parametros>}? )
                |   $id = classe # nula
                |   $id . metodo( {<parametros>}? )
<parametros>  ::=   <parametro> {, <parametros>? }
<parametro>   ::=   tipo_interno {<gerador>} ?
                |   $id
<gerador>     ::=   [maiuscula ; minuscula]
                |   [Classe]
<valores>     ::=   <valor> {, <valor>} ?
<valor>       ::=   inteiro
              ::=   real
              ::=   logico
              ::=   cadeia
```

Figura 5.2: Sintaxe do cromossomo (adaptada de (TONELLA, 2004)).

Cada <ação> pode tanto construir um novo objeto, determinar uma variável de cromossomo, indicada como **$id**, ou pode resultar numa chamada de método ou em um identificador de objeto identificado por **$id**. Um caso especial de determinação de objeto envolve o valor **null**, correspondendo a uma referência que não aponta para um objeto (não há alocação do objeto). O valor **null** é precedido pela classe **$id** (separada pelo caracter '#'), por isso esta variável pode ser utilizada como um parâmetro somente quando os tipos corresponderem (**null** não possui tipo de informação associado).

A segunda parte do cromossomo contém valores de entrada, que pertencem aos tipos pertencentes a linguagem de programação, e assim possuem a forma de um inteiro, real, lógica ou cadeia.

Segundo Tonella (2004), nem todos os cromossomos que são construídos de acordo com as regras da Figura 5.2 são bem formados. Um cromossomo é bem formado se:

1 Variáveis cromossômicas (indicadas como $id) estão sempre determinadas antes de serem utilizadas (como parâmetros ou chamadas de objetos ou métodos).

2 Nas ações para cada tipo integrante da linguagem, um valor de entrada é associado ao tipo de ocorrência na segunda metade do cromossomo, numa posição correspondente.

A ferramenta implementada garante que todos os cromossomos sejam bem formados, pois as variáveis indicadas como $id são sempre determinadas antes de serem utilizadas e havendo valores, estes sempre são associados às ações correspondentes.

Utilizando o programa TriTyp (Figura A.15.3) a representação do cromossomo fica como mostra a Tabela 5.1. Este programa avalia uma sequência de três números inteiros e produz como saída a classificação da entrada: se os números formam um triangulo equilátero, isósceles, escaleno ou não formam um triângulo.

Tabela 5.1 Exemplo de população.

$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 1, 3, 5
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 5, 5, 5
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 2, 3, 2,
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 3, 6, 7
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 4, 10 ,8
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 7, 7, 7
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 4, 1, 4
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 2, 5, 6
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 2, 2, 1
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 8, 6, 7

A Tabela 5.1 exemplifica uma população formada por 10 indivíduos utilizando a gramática descrita acima, na qual o objeto t é uma instanciação da classe TriTyp e é representado por $t=TriTyp(). Como o código TriTyp necessita de três números para verificar se estes formam um triângulo estes são passados através das chamadas dos métodos setI(int), setJ(int) e SetK(int) que são representados respectivamente por: $t.setI(int), $t.setJ(int) e $t.setK(int). A segunda metade do cromossomo contém os valores de entrada para os métodos setI, setJ e setK.

```
/*01*/ import java.io.*;
/*02*/ import java.util.*;
/*03*/ import java.io.Serializable;
/*04*/
/*05*/ public class TriTyp implements
       Serializable
/*06*/ {
/*07*/   public int i, j, k;
/*08*/   public int trityp;
/*09*/   public static int SCALENE = 1,
                    ISOSCELES = 2,
                    EQUILATERAL = 3,
                    NOT_A_TRIANGLE = 4;
/*10*/
/*11*/   public void setI(int v) {
/*12*/         i=v;
/*13*/   }
/*14*/
/*15*/   public void setJ(int v) {
/*16*/         j=v;
/*17*/   }
/*18*/
/*19*/   public void setK(int v) {
/*20*/        k=v;
/*21*/   }
/*22*/
/*23*/   / *@return
/*24*/              1 if scalene;
/*25*/              2 if isosceles;
/*26*/              3 if equilateral;
/*27*/              4 if not a triangle
/*28*/   */
/*29*/   public void type() {
/*30*/              if (i==j) { trityp=trityp+1; }
/*31*/              if (i==k) { trityp=trityp+2; }
/*32*/              if (j==k) { trityp=trityp+3; }
/*33*/
/*34*/              if (i<=0 || j<=0 || k<=0) {
/*35*/                 trityp=4;
/*36*/                 return;
/*37*/              }
/*38*/              if (trityp==0)
/*39*/              {
/*40*/                 if (i+j<=k ||
                          j+k<=i ||
                          i+k<=j) {
/*41*/                    trityp=4;
/*42*/                    return;
/*43*/              } else {
/*44*/                 trityp=1;
/*45*/                 return;
/*46*/              }
/*47*/        }
/*48*/        if (trityp>3) {
/*49*/           trityp=3;
/*50*/        } else if (trityp==1 && i+j>k)
/*51*/        {
/*52*/           trityp=2;
/*53*/         } else if (trityp==2 && i+k>j)
/*54*/         {
/*55*/            trityp=2;
/*56*/         } else if (trityp==3 && j+k>i)
/*57*/         {
/*58*/            trityp=2;
/*59*/         } else {
/*60*/            trityp=4;
/*61*/         }
/*62*/    }
/*63*/
/*64*/   public boolean equals(Object o)
/*65*/   {
/*66*/     if (!(o instanceof TriTyp)) return false;
/*67*/       TriTyp t=(TriTyp) o;
/*68*/       return (i==t.i && j==t.j
                     && k==t.k
                      && trityp==t.trityp);
/*69*/   }
/*70*/
/*71*/     public String toString()
/*72*/    {
/*73*/        String s= i + "-" + j + "-" + k + ": "
                       + trityp;
/*74*/       return s;
/*75*/     }
/*76*/  }
```

Figura 5.3: Exemplo adaptado de Vincenzi (VINCENZI et al., 2003).

5.2 Avaliação

O módulo Avaliação, como o nome diz, é responsável pela avaliação de cada indivíduo usando o módulo correspondente da ferramenta JaBUTi. Este módulo calcula a

cobertura de cada indivíduo. Cada indivíduo é convertido em uma entrada (caso de teste) para o programa que está sendo testado.

A ferramenta de teste JaBUTi gera em arquivo todos os elementos requeridos cobertos pela execução de um dado teste. A cada execução, para cada indivíduo da população, são obtidos o número de elementos cobertos por cada indivíduo. Essas informações de cobertura são armazenadas em um arquivo (ver Figura 5.4) de forma a representar uma tabela de cobertura de elementos por indivíduo.

Neste arquivo, cada coluna representa o elemento requerido e as linhas os indivíduos da população. O valor de 'X' representa que o elemento requerido foi coberto e '-' que não foi coberto. Os números no início de cada linha representam o número do caso de teste.

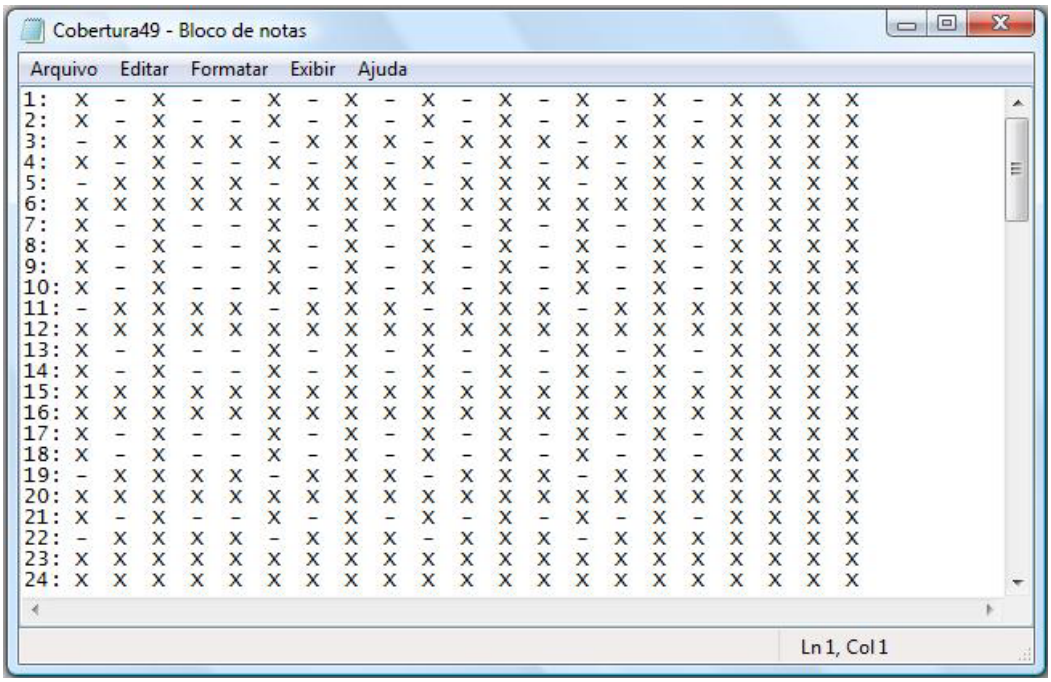


Figura 5.4: Ilustração do arquivo de cobertura dos indivíduos.

5.3 Evolução

Este módulo é responsável pelo processo de evolução e aplicação dos operadores genéticos. Dois mecanismos foram implementados um baseado no *fitness* e outro baseado numa outra função chamada ineditismo para considerar características específicas do contexto de teste de software.

A escolha do mecanismo de seleção é realizada no arquivo de configuração. A evolução termina quando o número de gerações atinge o número de gerações passado no arquivo de configuração. Os dois mecanismos implementados são:

1. ***Fitness:*** o *fitness* do indivíduo é calculado usando a matriz da Figura 5.4 e é dado pela Fórmula 5.1.

$$Fitness_x = \frac{\text{nro\_elem\_cobertos}}{\text{nro\_elem\_requeridos}} \qquad (5.1)$$

Na fórmula, o nro_elem_cobertos é o número de elementos cobertos do critério desejado e o nro_elem_requeridos é o número de total de elementos requeridos para determinado critério. Este objetivo varia de acordo com o critério escolhido. Quanto maior for o escore de fitness, pode-se interpretar que o indivíduo resolve bem o problema em questão.

Por exemplo, como ilustrado no Apêndice A, o caso de teste 1 cobre 14 de 45 elementos requeridos para a classe type, de acordo com a Fórmula 5.1, o *fitness* para o indivíduo será de 0,31.

$$Fitness_1 = \frac{14}{45} = 0{,}31$$

Baseado nesse *fitness* serão escolhidos os melhores indivíduos da população atual para gerar novos indivíduos que formarão a nova população.

2. **Ineditismo** garante que indivíduos que satisfaçam elementos inéditos, ou seja, ainda não cobertos por nenhum outro dado de teste, não sejam perdidos durante o processo de evolução. Um bônus de ineditismo é dado ao indivíduo que cobre sozinho um elemento requerido, como pode ser visto na Fórmula 5.2.

$$BonusIneditismoElemento_x = 100 \times \left(1 - \frac{nro\_indivíduos\_cobrem_x}{nro\_total\_individuos}\right) \qquad (5.2)$$

Onde o nro_individuos_cobrem$_x$ representa o número de indivíduos que cobrem o elemento em questão e nro_total_individuos o número total de indivíduos, ou seja, o tamanho da população.

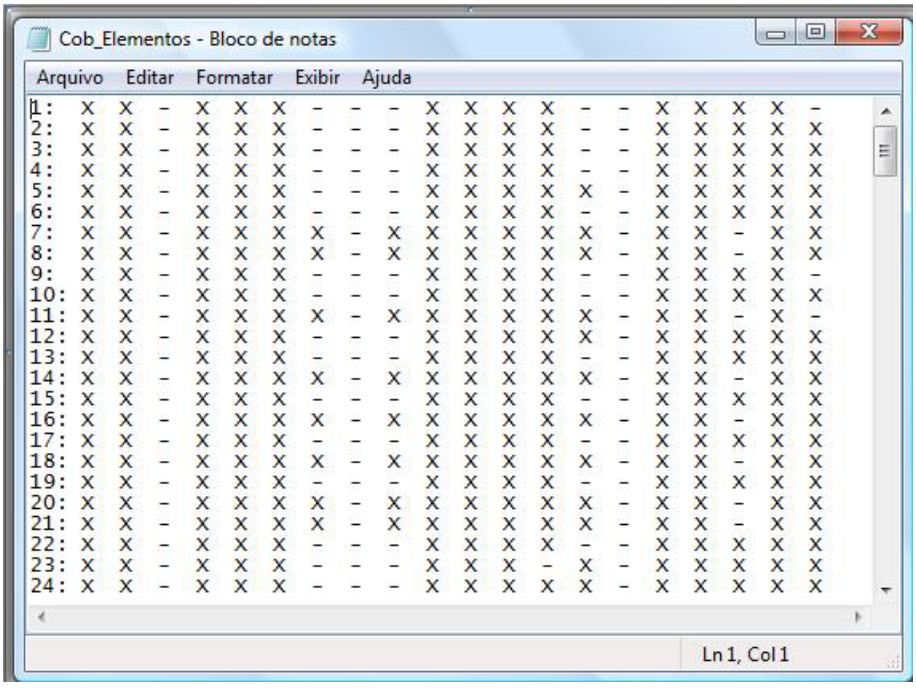


Figura 5.5: Ilustração do arquivo de cobertura de elementos.

A Figura 5.5 ilustra a matriz transposta dos elementos cobertos por cada caso de teste, no qual cada elemento tem-se uma linha de cobertura sendo que o n-ésimo caracter dessa linha tem o valor de 'X' se o n-ésimo indivíduo satisfez o elemento requerido em questão, '-' caso contrário.

Por exemplo, se tivermos uma população de tamanho 20 e pegarmos quantos indivíduos cobrem o elemento requerido '1' (ver Figura 5.5), o bônus de ineditismo para este elemento será 35, de acordo com a Fórmula 5.2:

$$BonusIneditismoElemento_1 = 100 \times \left(1 - \frac{13}{20}\right)$$

$$BonusIneditismoElemento_1 = 100 \times (1 - 0.65)$$

$$BonusIneditismoElemento_1 = 100 \times 0.35$$

$$BonusIneditismoElemento_1 = 35$$

O cálculo do bônus de ineditismo do indivíduo é ilustrado pela Fórmula 5.3:

$$BonusIneditismoIndivíduo_y = \sum_{i=0}^{q} BonusIneditismoElemento_x \qquad (5.3)$$

Onde o $BonusIneditismoElemento_x$ é o bônus de cada elemento x coberto pelo indivíduo y e q a quantidade de elementos requeridos.

Utiliza-se um método de seleção para obter o(s) indivíduo(s) para sofrer(em) a ação dos operadores genéticos para gerar novo(s) indivíduo(s) da próxima geração, sendo este

processo repetido até que uma população de mesmo tamanho seja gerada. Os métodos de seleção implementados são:

- Seleção por Roleta: faz o somatório do valor de *fitness* de cada indivíduo, em seguida um valor que varia entre 0 até somatório é sorteado aleatoriamente. Será selecionado o indivíduo que ao somar seu valor de *fitness* na variável somatória, supera ou se iguala ao valor sorteado.
- Seleção por Torneio: escolhe-se aleatoriamente 2 indivíduos sendo selecionado aquele que tiver o maior valor de *fitness*.

Com o conjunto inicial de indivíduos estabelecido, são utilizados os operadores genéticos até se obter a melhor solução. Para este trabalho foram escolhidos os operadores de mutação. A seguir são descritos o conjunto de operadores de mutação que podem ser utilizados para teste evolucionário de classes (TONELLA, 2004).

- **Mutação no valor de entrada**: o valor é substituído por um valor randômico do mesmo tipo.

  Exemplo:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 6, 3, 8

- **Mudança de construtor**: um dos construtores é randomicamente modificado. Valores ou objetos previamente requeridos (não utilizados) são perdidos. Novos valores ou objetos são inseridos. Objetos já existentes podem ser reutilizados.

  Exemplo:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $teste=Teste(): $teste.setI(int) : $teste.setJ(int) : $teste.setK(int) : $teste.type() @ 9, 3, 8

- **Inserção de uma invocação de método:** uma nova invocação de método é inserida. Valores ou parâmetros necessários para a invocação são inseridos.

  Exemplo:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $t=TriTyp() : $t.toString() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

- **Remoção de uma invocação de método:** algumas invocações de métodos são aleatoriamente selecionadas e removidas. Valores ou objetos requeridos anteriormente são excluídos.

  Exemplo:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $t=TriTyp() : $t.setJ(int) : $t.setK(int) : $t.type() @ 3, 8

- **Cruzamento ponto único**: depois de selecionados dois cromossomos, escolhe-se aleatoriamente um ponto para o corte.

  Pais:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $s=TriTyp() : $s.setI(int) : $s.setJ(int) : $s.setK(int) : $s.type() @ 4, 1, 4

  Filhos:

  $t=TriTyp() : $t.setI(int) : *$s.setJ(int) : $s.setK(int) : $s.type()* @ **9**, *1, 4*

  $s=TriTyp() : $s.setI(int) : **t.setJ(int) : $t.setK(int) : $t.type()** @ *4,* **3**, **8**

- **Cruzamento troca de parâmetros**: depois de selecionados dois cromossomos, escolhe-se aleatoriamente um parâmetro de cada cromossomo pai e faz-se a mudança destes parâmetros para geração de cromossomos filhos.

  Pais:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 4, 1, 4

  Filhos:

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @1, 3, 8

  $t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 4, 9, 4

Existe implementado o **elitismo**, esta técnica introduz indivíduos na próxima geração baseado em uma lista ordenada segundo o valor de *fitness*, assegurando que bons indivíduos com *fitness* altos estejam na próxima geração.

5.4 Início

Este módulo é responsável por receber a configuração inicial (Arquivo de Configuração) fornecido pelo testador. Os parâmetros passados no Arquivo de Configuração devem seguir o seguinte critério:

- Um parâmetro por linha;

- Cada declaração deve estar na forma “parâmetro: valor”;
- Qualquer linha que não siga este padrão ou que não contenha as palavras reservadas é ignorada.

Os parâmetros aceitos são apresentados na Tabela 5.2.

Tabela 5.2: Parâmetros do arquivo de configuração.

Parâmetros	
Arquivo Compilado	Tamanho do indivíduo
Arquivo de Teste	Tamanho da população
Criar Pasta com Nome	Taxa de mutação
Ineditismo	Taxa de *cruzamento*
Elitismo	Taxa de *cruzamento*2
Critério	Seleção
Número de Gerações	

Um exemplo de arquivo de configuração é apresentado na Figura 5.6. Percebe-se que ele contém informações que podem se dividir em quatro seções descritas a seguir:

5.4.1 Ferramenta de Teste

Esta seção inclui parâmetros correspondentes à ferramenta de teste, como: nome do arquivo compilado e o critério escolhido. O módulo Avaliação é responsável pelo controle da ferramenta. O script da ferramenta é executado pelo módulo Avaliação para gerar os elementos requeridos pelo critério. Contém os seguintes dados:

- Arquivo Compilado: arquivo a ser testado;
- Arquivo de Teste: arquivo que converte a população em uma JUnit para que possa ser utilizada pela JaBUTi;
- Critério: corresponde a um número que está relacionado com o critério que se deseja testar:
    1. Todos-nós-ei;
    2. Todos-nós-ed;
    3. Todos ramos-ei;
    4. Todos ramos-ed;
    5. Todos-usos-ei;
    6. Todos-usos-eid
    7. Todos-Pot-usos-ei
    8. Todos-Pot-usos-ed

5.4.2 Entradas

Esta seção é relacionada com a execução do programa testado P. É usada para criar a população inicial e contém:

- Tamanho do indivíduo: corresponde à quantidade de chamada de métodos que será feita em cada caso de teste;
- Tamanho da população: corresponde ao número de indivíduos que serão gerados.

5.4.3 Estratégias de Evolução

Esta seção é relacionada com o processo de evolução e será utilizado pelo módulo Evolução. Contém:

- Taxa de mutação: determina a taxa de quantos indivíduos sofrerão mutação na população;
- Taxa de *cruzamento*: determina a taxa de quantos indivíduos sofrerão *cruzamento* de ponto único na população;
- Taxa de *cruzamento2*: determina a taxa de quantos indivíduos sofrerão *cruzamento2* com troca de parâmetros na população;
- Número de Gerações: determina o total de gerações a serem executadas;
- Ineditismo: guarda o(s) único(s) indivíduo(s) que cobriram determinado elemento requerido;
- Elitismo: guarda os indivíduos que contém os melhores *fitness* para que sejam utilizados na nova população;
- Seleção: determina qual será o método utilizado para selecionar os indivíduos que participarão nos operadores de mutação.
    0. Nenhum (utilizado para a geração aleatória)
    1. Roleta
    2. Torneio

Arquivo Compilado: TriTyp.class
Arquivo de Teste: TTeste.java
Criar Pasta com Nome: TriTyp
Ineditismo: Sim
Elitismo: Sim
Critério: 1
Tamanho do Indivíduo: 50
Tamanho da População: 100
Número de Gerações: 100
Taxa de Mutação: 0.75
Taxa de Cruzamento: 0.75
Taxa de Cruzamento2: 0.75
Seleção: 1

Figura 5.6: Arquivo de configuração.

5.5 Aspectos de Implementação

A ferramenta foi implementada em linguagem Java para plataforma Windows. Esta ferramenta faz integração com a ferramenta JaBUTi descrita no Capítulo 4. A integração possibilita a aplicação dos critérios de teste implementados pela JaBUTi na geração de dados de teste. Os critérios são: Todos-Nós; Todos-Ramos; Todos-Usos e Todos-Potenciais-Usos.

A Figura 5.7 mostra o pseudocódigo do algoritmo implementado. O primeiro passo é ler o arquivo de configuração que contém os dados necessários para a geração dos dados de teste. Em seguida é feita a criação do projeto da classe Java que será testada utilizando a JaBUTi, que originará um relatório contendo informações sobre a classe em teste. Através deste relatório é possível determinar o conjunto de elementos requeridos para o critério escolhido, caso este conjunto não seja vazio, um conjunto de dados de teste é gerado aleatoriamente (população). O algoritmo gera novos dados de teste enquanto o número de gerações não for atingido. Dados de teste na população atual são executados utilizando a JaBUTi, um novo relatório é gerado e através deste é possível verificar a cobertura dos elementos requeridos. Com esta cobertura é possível avaliar se o caso de teste foi o único a

cobrir determinado elemento requerido (ineditismo). Caso o ineditismo seja escolhido no arquivo de configuração, os casos de teste que estão contidos no vetor ineditismo ganham um bônus no cálculo da medida de proximidade *fitness*. Uma nova população é criada através dos operadores de mutação para substituir a população atual.

```
        Algoritmo de Geração de Dados (ClasseTeste :  Class, Critério C)
        //Módulo Início
1       lê arquivo de configuração
        //Ferramenta de Teste
2       criar projeto {java -cp Jabuti-bin.zip: br.jabuti.cmdtool.CreateProject -b <classe> -p
           <projeto.jbt>}
3       gera relatório utilizando ferramenta JaBUTi
4       elementos_requeridos[] ← elementos_requeridos(ClasseTeste, C)
        //Módulo Geração da População
5       população ← geração_da_população(tamanho) {tamanho passado pelo Arquivo de
                                                    Configuração}
6       enqto geração < numero_gerações {numero_gerações passado pelo Arquivo de
                                                    Configuração}
            //Módulo Avaliação
            //Ferramenta de Teste
7           codificar população (criar JUnit)
8           criar projeto {java -cp Jabuti-bin.zip: br.jabuti.cmdtool.CreateProject -b <classe>
                                    -p <projeto.jbt>}
9           instrumentar projeto {java -cp Jabuti-bin.zip br.jabuti.probe.ProberInstrum -o
                                    <arquivo.jar> -p <projeto.jbt> <classe a executar>}
10          importar casos de teste {java -cp Jabuti-bin.zip br.jabuti.cmdtool.ImportTestCase
                                    -p <projeto.jbt> }
11          gerar relatório {java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p
                                    <projeto.jbt> -me -o metodos.xml}
            //Fim Ferramenta de Teste
12          lê relatório para ver a cobertura
13          se (ineditismo == "sim")
14              verifica inedistismo
15          fim se
            //Módulo Evolução
16          calcula fitness
17          se (elitismo == "sim")
18              aplica elitismo
19          aplicar operadores de mutação
20          populacao ← nova_população
21          geracao ← geracao +1;
22      fim enqto
```

Figura 5.7: Pseudocódigo implementado.

É necessário que um novo projeto seja gerado sempre que for feita a avaliação da população, pois toda vez que são executados os casos de teste a informação sobre a cobertura fica armazenada no projeto. Por exemplo, se tivermos uma população com tamanho igual a 50, realizarmos a avaliação desta utilizando a ferramenta JaBUTi, coletarmos esses dados para aplicação dos operadores genéticos para que uma nova população seja obtida, na segunda rodada do algoritmo, se não criarmos um novo projeto constará que este terá um conjunto com 100 casos de teste, pois as informações da avaliação anterior (50 casos de teste) estão contidas no projeto e somarão aos 50 novos que estarão sendo avaliados.

5.6 Exemplo

A Figura 5.8 ilustra um exemplo de população para TriTyp, sendo o tamanho do indivíduo igual a 10. Cada linha representa um caso de teste, sendo que o número da linha corresponde ao número do caso de teste. Após a criação da população, é necessário converter a população para o formato JUnit (ver Figura 5.9), para executar os testes utilizando a JaBUTi.

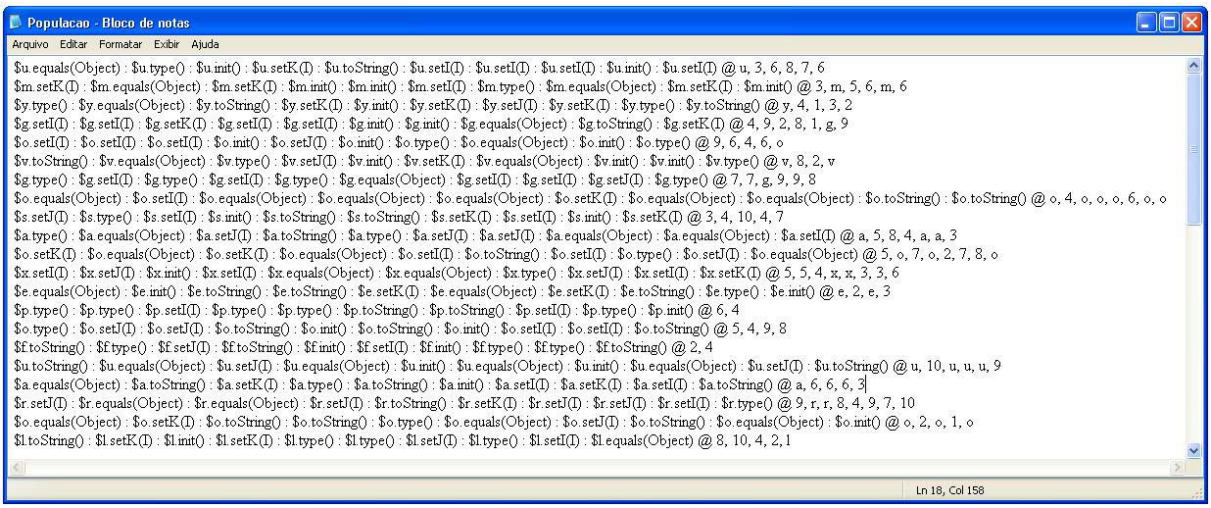


Figura 5.8: População criada pela ferramenta TDSGen/OO.

Depois de executados os testes, gera-se o relatório de métodos (ver Figura 5.10) para verificar a cobertura dos dados de teste e com este calcular a função de *fitness*. Um arquivo chamado registro é gerado (Figura 5.11), este contém o número de elementos requeridos e elementos cobertos da classe e seus métodos. Após o calculo do *fitness* são aplicados os operadores genéticos, e uma nova população é obtida.

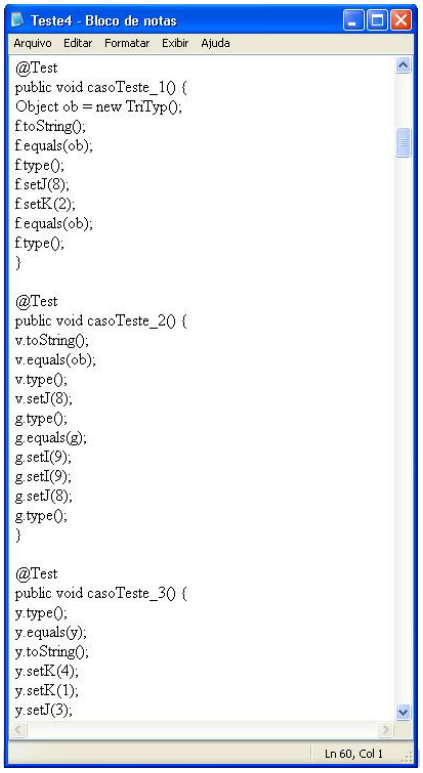


Figura 5.9: JUnit gerada a partir da população.


```
- <method class_name="TriTyp" full_name="TriTyp.type()V" name="type()V">
  - <coverage>
      <All-Nodes-ei covered="22" percentage="75.0" required="29" />
      <All-Nodes-ed covered="0" percentage="0.0" required="0" />
      <All-Edges-ei covered="27" percentage="64.0" required="42" />
      <All-Edges-ed covered="0" percentage="0.0" required="0" />
      <All-Uses-ei covered="59" percentage="44.0" required="133" />
      <All-Uses-ed covered="0" percentage="0.0" required="0" />
      <All-Pot-Uses-ei covered="47" percentage="27.0" required="168" />
      <All-Pot-Uses-ed covered="0" percentage="0.0" required="0" />
    </coverage>
  - <requirements>
    - <All-Nodes-ei>
      - <uncovered>
          <requirement weight="0">129</requirement>
          <requirement weight="0">151</requirement>
          <requirement weight="0">181</requirement>
          <requirement weight="0">197</requirement>
          <requirement weight="0">213</requirement>
          <requirement weight="0">229</requirement>
          <requirement weight="0">269</requirement>
        </uncovered>
      - <covered>
          <requirement effectivetcs="0029 0028 0026 0025 0024 0023 0022 0021 0020 0050 0019 0018 0017 0049 0016 0048 0015 0014 0046 0013 0045 0012 0044 0011 0043
            0010 0042 0041 0040 0009 0008 0007 0039 0006 0038 0005 0037 0036 0004 0035 0003 0034 0002 0033 0001 0032 0031 0030">0</requirement>
          <requirement effectivetcs="0029 0025 0023 0022 0021 0020 0050 0019 0018 0049 0016 0048 0015 0014 0046 0013 0045 0012 0044 0011 0043 0010 0041 0040 0008
            0007 0006 0005 0036 0004 0035 0003 0034 0002 0001 0032 0031 0030">11</requirement>
          <requirement effectivetcs="0028 0024">113</requirement>
          <requirement effectivetcs="0028 0024">145</requirement>
          <requirement effectivetcs="0028 0024 0039">157</requirement>
          <requirement effectivetcs="0028 0024 0039">165</requirement>
          <requirement effectivetcs="0028">173</requirement>
          <requirement effectivetcs="0028">205</requirement>
          <requirement effectivetcs="0029 0028 0026 0025 0024 0023 0022 0021 0020 0050 0019 0018 0017 0049 0016 0048 0015 0014 0046 0013 0045 0012 0044 0011 0043
            0010 0042 0041 0040 0009 0008 0007 0039 0006 0038 0005 0037 0036 0004 0035 0003 0034 0002 0033 0001 0032 0031 0030">21</requirement>
          <requirement effectivetcs="0028">237</requirement>
          <requirement effectivetcs="0028">245</requirement>
          <requirement effectivetcs="0028">261</requirement>
          <requirement effectivetcs="0028 0024 0039">274</requirement>
          <requirement effectivetcs="0029 0025 0023 0022 0021 0020 0019 0018 0049 0016 0048 0015 0014 0046 0013 0045 0012 0044 0011 0043 0010 0041 0040 0009 0008
            0007 0006 0005 0037 0036 0004 0035 0003 0034 0002 0001 0032 0031 0030">32</requirement>
          <requirement effectivetcs="0029 0028 0026 0025 0024 0023 0022 0021 0020 0050 0019 0018 0017 0049 0016 0048 0015 0014 0046 0013 0045 0012 0044 0011 0043
```


Figura 5.10: Exemplo de relatório de métodos.

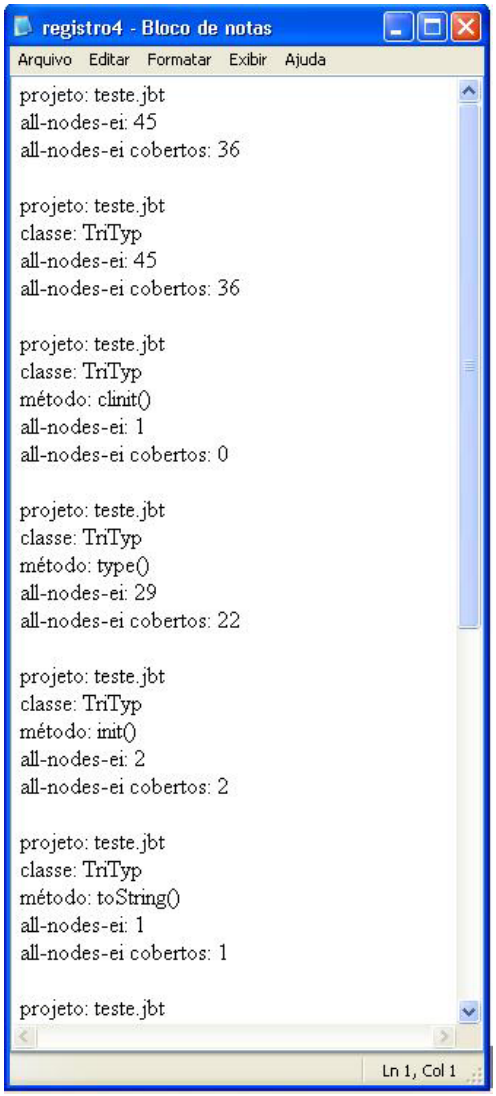


Figura 5.11: Arquivo gerado pela ferramenta TDSGen/OO.

5.7 Considerações Finais

Neste capítulo foi descrita a ferramenta TDSGen/OO que permite a integração com a ferramenta JaBUTi e a utilização de critérios de teste implementados por esta.

No próximo capítulo são descritos os resultados de um experimento para validar a implementação da ferramenta TDSGen/OO e também das estratégias implementadas.

6 EXPERIMENTO DE VALIDAÇÃO

O experimento descrito neste capítulo, tem por objetivo validar a implementação da ferramenta TDSGen/OO e verificar o desempenho do AG considerando as estratégias implementadas e descritas no Capítulo 5.

6.1 Descrição

Os programas utilizados no experimento estão descritos Tabela 6.1[5]. Os programas apresentados foram executados com os critérios implementados pela ferramenta de teste JaBUTi: Todos-Nós; Todos-Ramos; Todos-Usos e Todos-Potenciais-Usos.

Tabela 6.1: Programas do experimento de validação (POLO et al., 2009).

Programa	Descrição
TriTyp	Possui como entrada três números inteiros positivos, verificando se estes formam um triângulo equilátero, escaleno, isósceles ou se não forma um triângulo.
BubCorrecto	Possui como entrada números inteiros positivos, sendo estes são colocados em um vetor e ordenados (ordem crescente).
Bisect	Possui como entrada três números reais. Determina raiz de uma função utilizando o método da Bissecção.
Mid	Possui como entrada três números inteiros e os classifica como maior, menor e o número médio.

Para obter uma base comparativa, cada programa citado na Tabela 6.1 foi testado como segue:

- 5 execuções com geração de dados aleatória para três configurações diferentes;
- 5 execuções com AG para nove configurações diferentes;
- 5 execuções com AG Ineditismo para nove configurações diferentes;

[5] Disponíveis no site: http://www.inf-cr.uclm.es/www/mpolo/stvr/

No experimento foram comparadas três estratégias. Todas elas foram utilizadas para o teste de todos métodos de cada programa e todos os critérios implementados pela ferramenta JaBUTi.

1. A primeira estratégia, a geração aleatória foi obtida setando o número de gerações do algoritmo para zero. Para esta estratégia foram realizados experimentos considerando três tamanhos de população: 50, 100 e 200. Conforme configurações da Figura 6.1.

Arquivo Compilado: TriTyp.class
Arquivo de Teste: TTeste.java
Criar Pasta com Nome: TriTyp
Ineditismo: Não
Elitismo: Não
Critério: 1
Tamanho do Indivíduo: 20
Tamanho da População: 50, 100, 200
Número de Gerações: 0
Taxa de Mutação: 0
Taxa de Cruzamento: 0
Taxa de Cruzamento2: 0
Seleção: 0

Figura 6.1: Configurações para Geração Aleatória.

2. A segunda estratégia foi a de um Algoritmo Genético (AG), tanto elitismo como o ineditismo não foram setados. Nesse caso também foram utilizados três tamanhos de população e três estratégias de gerações perfazendo três arquivos de configuração diferentes conforme Figura 6.2.

Arquivo Compilado: TriTyp.class	Arquivo Compilado: TriTyp.class	Arquivo Compilado: TriTyp.class
Arquivo de Teste: TTeste.java	Arquivo de Teste: TTeste.java	Arquivo de Teste: TTeste.java
Criar Pasta com Nome: TriTyp	Criar Pasta com Nome: TriTyp	Criar Pasta com Nome: TriTyp
Ineditismo: Não	Ineditismo: Não	Ineditismo: Não
Elitismo: Não	Elitismo: Não	Elitismo: Não
Critério: 1	Critério: 3	Critério: 7
Tamanho do Indivíduo: 20	Tamanho do Indivíduo: 20	Tamanho do Indivíduo: 20
Tamanho da População: 50, 100, 200	Tamanho da População: 50, 100, 200	Tamanho da População: 50, 100, 200
Número de Gerações: 50	Número de Gerações: 100	Número de Gerações: 200
Taxa de Mutação: 0.75	Taxa de Mutação: 0.75	Taxa de Mutação: 0.75
Taxa de Cruzamento: 0.75	Taxa de Cruzamento: 0.75	Taxa de Cruzamento: 0.75
Taxa de Cruzamento2: 0.75	Taxa de Cruzamento2: 0.75	Taxa de Cruzamento2: 0.75
Seleção: 1	Seleção: 1	Seleção: 1

Figura 6.2: Configurações para Algoritmo Genético.

3. A terceira estratégia foi identificada como AG Ineditismo (AGI), utilizando tanto elitismo como ineditismo foram setados como verdadeiros. Nesse caso também foram utilizados três tamanhos de população 50, 100 e 200, com três números de gerações 50, 100 e 200 ilustrados na Figura 6.3.

Arquivo Compilado: TriTyp.class	Arquivo Compilado: TriTyp.class	Arquivo Compilado: TriTyp.class
Arquivo de Teste: TTeste.java	Arquivo de Teste: TTeste.java	Arquivo de Teste: TTeste.java
Criar Pasta com Nome: TriTyp	Criar Pasta com Nome: TriTyp	Criar Pasta com Nome: TriTyp
Ineditismo: Sim	Ineditismo: Sim	Ineditismo: Sim
Elitismo: Sim	Elitismo: Sim	Elitismo: Sim
Critério: 1	Critério: 3	Critério: 7
Tamanho do Indivíduo: 20	Tamanho do Indivíduo: 20	Tamanho do Indivíduo: 20
Tamanho da População: 50, 100, 200	Tamanho da População: 50, 100, 200	Tamanho da População: 50, 100, 200
Número de Gerações: 50	Número de Gerações: 100	Número de Gerações: 200
Taxa de Mutação: 0.75	Taxa de Mutação: 0.75	Taxa de Mutação: 0.75
Taxa de Cruzamento: 0.75	Taxa de Cruzamento: 0.75	Taxa de Cruzamento: 0.75
Taxa de Cruzamento2: 0.75	Taxa de Cruzamento2: 0.75	Taxa de Cruzamento2: 0.75
Seleção: 1	Seleção: 1	Seleção: 1

Figura 6.3: Configurações para AGI.

Em todas as estratégias e configurações o tamanho do indivíduo ficou fixo em 20. Os resultados obtidos para cada configuração e programa considerando cada estratégia encontram-se no Apêndice C, contendo também os valores dos desvios padrão para cada programa e critério.

6.2 Resultados

Para contabilizar os resultados de cada estratégia, para cada programa, foram tomadas as médias de cinco execuções para cada critério (todos-nós, todos-ramos, todos-usos e todos-potenciais-usos) do algoritmo correspondente.

Os resultados são resumidos calculando-se a média de todos os programas. Esses resultados são apresentados nas Tabelas 6.2 a 6.4. Pode haver variação nos valores obtidos para cada critério, pois alguns programas tiveram uma boa cobertura e outros não tão boas.

Tabela 6.2: Média de cobertura para 50 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	52%	42%	39%	36%
	100	58%	47%	43%	42%
	200	60%	49%	48%	29%
AG	50	60%	46%	42%	39%
	100	57%	41%	39%	39%
	200	59%	44%	41%	41%
AG Ineditismo	50	70%	57%	55%	48%
	100	78%	68%	64%	61%
	200	84%	75%	75%	68%

A cobertura dos programas testados foi melhor quando o tamanho da população foi igual a 200, para todas as gerações (50, 100 e 200), sendo às vezes igualada à de tamanho 100. Dentre os algoritmos testados, o que obteve melhor cobertura foi o programa Mid, já o que teve pior média de cobertura foi o Bisect. Observando a Tabela 6.3 nota-se uma grande diferença nos resultados do critério Todos-Potenciais-Usos com a geração aleatória, isso se deve à baixa cobertura do programa BubCorrecto para este critério, o mesmo ocorre na Tabela 6.4 com o algoritmo genético.

Tabela 6.3: Média de cobertura para 100 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	58%	51%	46%	41%
	100	56%	43%	40%	35%
	200	58%	54%	53%	38%
AG	50	56%	41%	38%	39%
	100	50%	38%	36%	34%
	200	51%	37%	30%	29%
AG Ineditismo	50	72%	60%	60%	49%
	100	81%	69%	69%	59%
	200	85%	76%	75%	60%

Tabela 6.4: Média de cobertura para 200 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	59%	47%	38%	37%
	100	61%	49%	42%	38%
	200	62%	52%	46%	41%
AG	50	49%	32%	28%	29%
	100	68%	53%	45%	42%
	200	60%	44%	40%	40%
AG Ineditismo	50	77%	62%	55%	46%
	100	82%	62%	68%	52%
	200	84%	67%	73%	54%

Já as Tabelas 6.3 e 6.4 não apresentam grande variação nos resultados obtidos para os critérios avaliados. Os diferentes números de gerações podem ser visualizados nos gráficos das Figuras 6.4 a 6.6.

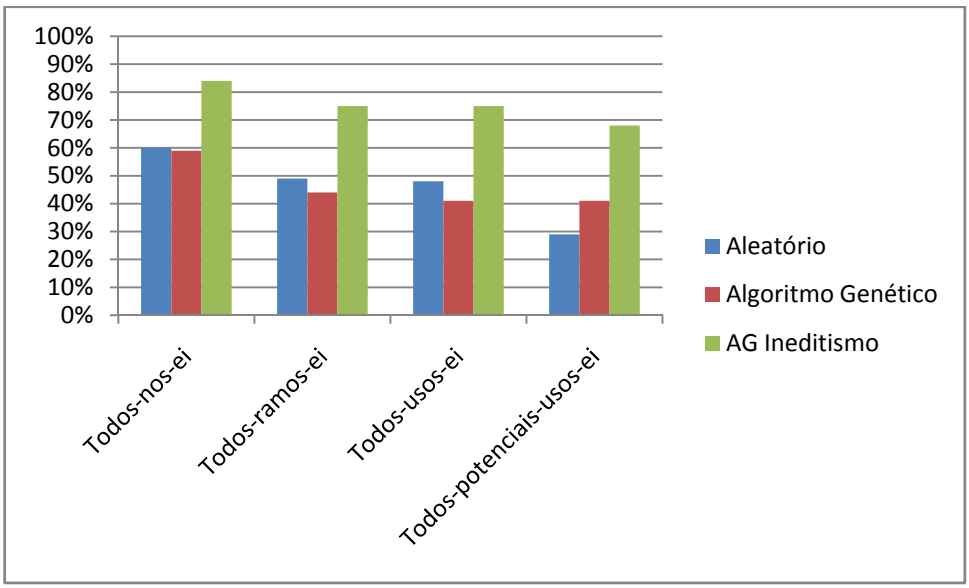


Figura 6.4: Gráfico para 50 gerações.

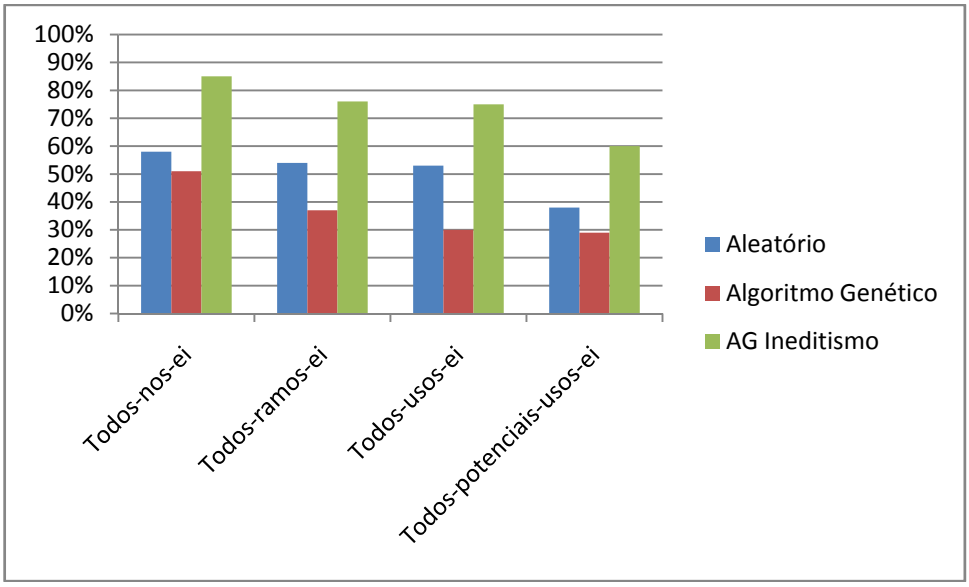


Figura 6.5: Gráfico para 100 gerações.

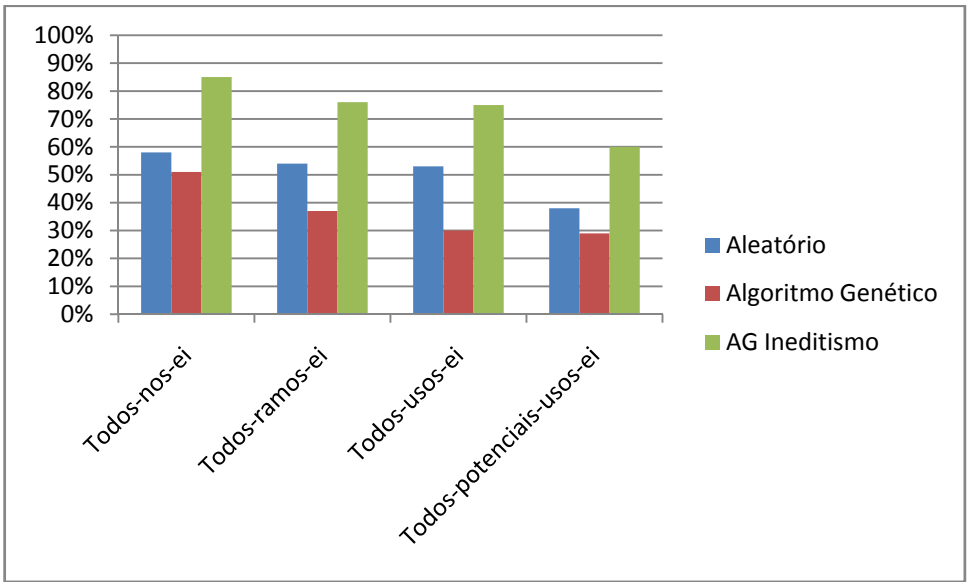


Figura 6.6: Gráfico para 200 gerações.

Outro resultado está relacionado aos tempos de execução. As Tabelas 6.5 a 6.7 mostram a média do tempo de execução das três estratégias considerando todos os programas e todos os critérios respectivamente para os três números de gerações utilizados.

Tabela 6.5: Média de tempo para 50 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	0:00:59	0:01:02	0:01:09	0:01:06
	100	0:01:10	0:01:08	0:01:11	0:01:07
	200	0:01:08	0:01:10	0:01:12	0:01:08
AG	50	0:57:54	0:56:54	0:56:12	0:58:41
	100	0:51:54	0:53:54	0:50:59	0:52:22
	200	0:52:47	0:51:23	0:52:23	0:53:13
AG Ineditismo	50	0:44:27	0:43:49	0:44:02	0:44:47
	100	0:47:30	0:45:29	0:47:57	0:46:58
	200	0:48:56	0:47:34	0:49:22	0:47:44

Tabela 6.6: Média de tempo para 100 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	0:01:04	0:01:00	0:01:08	0:01:09
	100	0:01:07	0:01:03	0:01:05	0:01:01
	200	0:01:08	0:01:10	0:01:13	0:01:12
AG	50	1:31:21	1:30:12	1:35:42	1:36:51
	100	1:54:43	1:52:31	1:50:34	1:45:23
	200	1:46:17	1:54:28	1:36:02	1:50:33
AG Ineditismo	50	1:26:50	1:26:34	1:16:02	1:36:15
	100	1:34:11	1:33:21	1:24:11	1:44:01
	200	1:55:49	1:45:49	1:40:49	1:50:56

Tabela 6.7: Média de tempo para 200 Gerações.

	Tamanho População	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	0:01:12	0:01:10	0:01:06	0:01:05
	100	0:01:11	0:01:08	0:01:11	0:01:02
	200	0:01:12	0:01:03	0:01:12	0:01:09
AG	50	3:42:02	3:41:02	3:42:02	3:40:22
	100	3:47:55	3:45:55	3:47:55	3:41:47
	200	3:31:14	3:34:14	3:31:14	3:32:43
AG Ineditismo	50	2:58:00	2:57:20	2:56:38	2:55:29
	100	3:09:33	3:02:33	3:09:33	3:10:45
	200	3:29:23	3:22:23	3:29:23	3:30:30

Eles também podem ser melhor visualizados nos gráficos das Figuras 6.7 a 6.9.

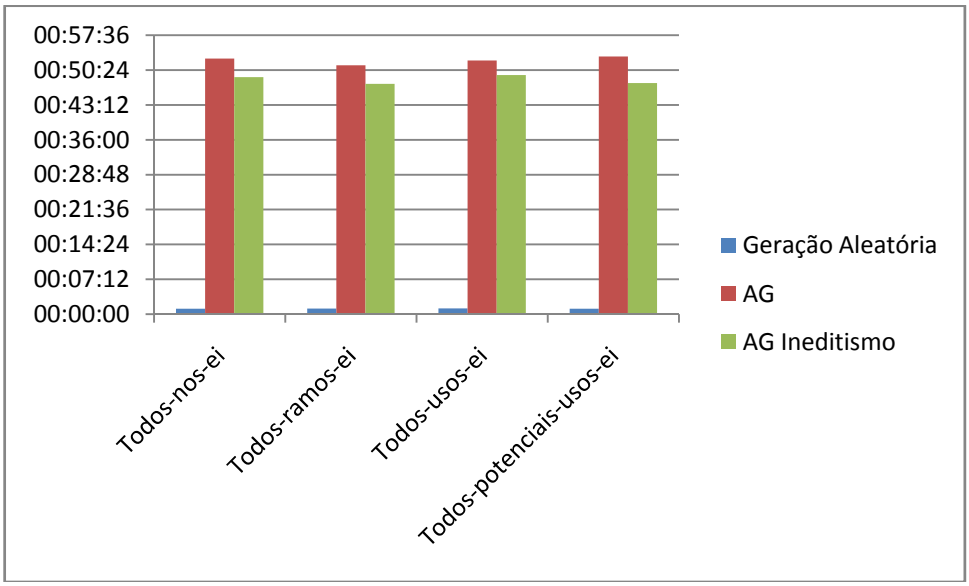


Figura 6.7: Gráfico de média de tempo para 50 Gerações.

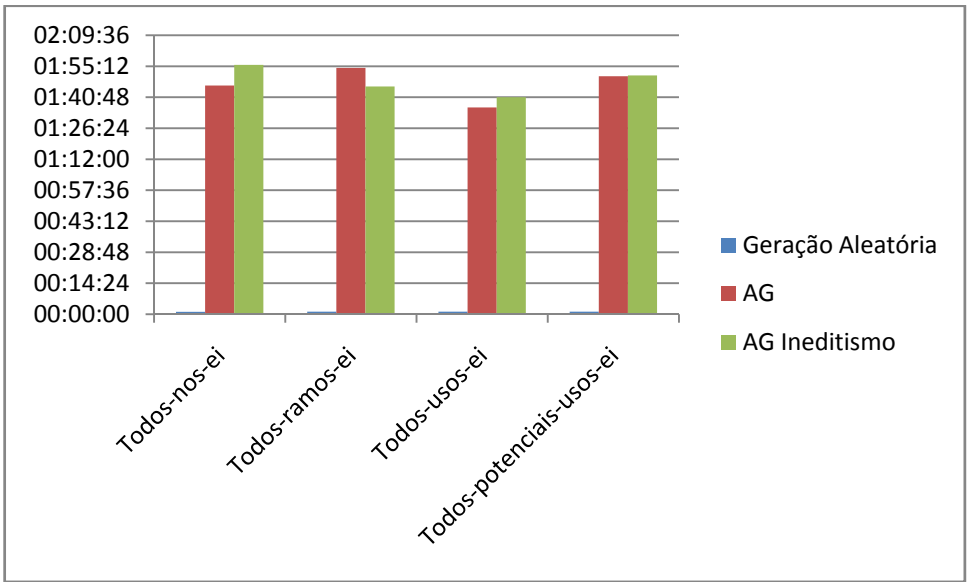


Figura 6.8: Gráfico de média de tempo para 100 Gerações.

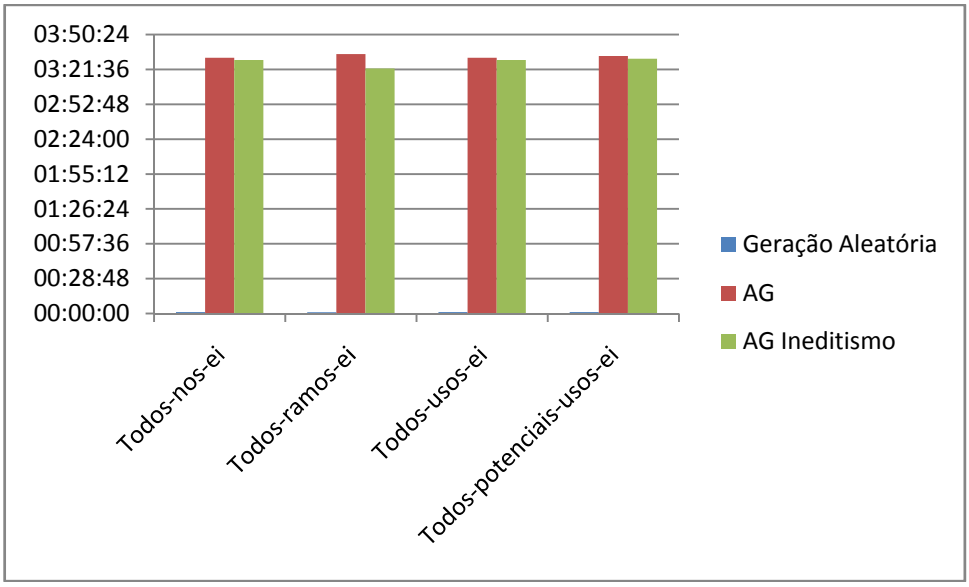


Figura 6.9: Gráfico de média de tempo para 200 Gerações.

6.3 Análise dos Resultados

Nessa seção os resultados são analisados de acordo com a cobertura e tempo de execução, e considerando a configuração dada pelo tamanho da população e número de geração.

1. Com relação a cobertura

Percebe-se que o AGI obteve sempre a melhor cobertura independentemente do número de gerações, seguido pela geração aleatória e do AG. Isto porque alguns bons indivíduos gerados inicialmente são perdidos no processo de evolução. Percebe-se que o AG começa a se igualar com a estratégia aleatória, ou seja, a se recuperar conforme o número de gerações aumenta, pois para 200 gerações, os resultados de ambas estratégias são mais similares.

Também pode se perceber que o desempenho do AGI também é ligeiramente melhor para um número de gerações maior. Outro ponto importante mostra a dificuldade de satisfação dos critérios, independentemente da estratégia, os critérios baseados em fluxo de dados são mais difíceis de satisfazer.

Outro ponto é que o tamanho da população também influencia, as melhores coberturas para as três estratégias foram alcançadas com um tamanho de população igual a 200.

Pelos gráficos das Figuras 6.4 a 6.6 é possível verificar que o AGI alcançou uma cobertura satisfatória, pois, em muitos casos, a existência de caminhos não executáveis não permite a total cobertura (100%) dos critérios.

Por exemplo, o programa BubCorrecto apresentou cobertura dos elementos requeridos pelo critério todos-usos-ei (Tabela C.7) de 26% com geração aleatória, 26% com o AG e 75% com o AGI. Determinando-se manualmente os elementos requeridos não executáveis, foram identificados 10% de associações não executáveis para esse critério. Portanto uma cobertura de 75% para esse critério é bastante razoável, pois em torno de apenas 15% de elementos não foram cobertos automaticamente.

2. Tempo de execução

O tempo de geração aumenta conforme o tamanho da população aumenta, pois mais indivíduos são avaliados. O mesmo acontece se o número de gerações aumenta. O tempo de execução da geração aleatória é muito baixo com relação à execução dos AGs como era esperado (40 a 50 vezes maior) dependendo do número de gerações. Quanto maior esse número, maior o tempo. Entretanto, para os experimentos realizados, a diferença medida em minutos foi considerada aceitável, por se tratarem de programas simples. Não se percebem diferenças significativas entre o tempo gasto com o AG e o AGI.

6.4 Considerações Finais

Os resultados apresentados neste capitulo mostram que as estratégias implementadas pela ferramenta TDSGen/OO aumentam o desempenho do AG, sem aumentar significativamente o custo.

O tempo de execução aumenta de acordo com o número de gerações, não tendo muita variação entre os critérios testados, isso se dá pela avaliação dos indivíduos para a nova geração.

O tamanho do indivíduo não foi avaliado neste experimento, mas deve influenciar e por isso ser considerado em trabalhos futuros.

7 CONCLUSÕES E TRABALHOS FUTUROS

O objetivo principal do teste de software é aumentar a confiabilidade e garantir a qualidade do software para que este tenha o mínimo de erros possível. Entretanto, essa atividade consome grande parte dos custos envolvidos durante o desenvolvimento e mesmo assim, não garante um produto completamente livre de erros.

A geração de dados de teste é uma das tarefas mais difíceis de ser automatizada, devido a questões de indecidibilidade e limitações inerentes à própria atividade de teste.

Além disso, levando em consideração o paradigma OO e a grande atenção e aceitação que vem sendo dada por parte dos pesquisadores de software a este paradigma, faz-se necessária a avaliação das ferramentas de teste para programas OO, bem como dos critérios de teste apoiados por estas ferramentas.

Para isso, este trabalho descreve uma ferramenta chamada TDSGen/OO, com o objetivo de gerar dados de teste para satisfazer os critérios baseados em *bytecode* implementados pela ferramenta JaBUTi. A idéia é oferecer um ambiente para a aplicação de todos os critérios implementados em uma estratégia de teste e assim, reduzir os custos e esforço gasto para aplicação dos critérios e, consequentemente, facilitar a atividade de teste de software desenvolvido com esse paradigma.

A ferramenta TDSGen/OO implementa um AG e um AG Ineditismo que considera características particulares com relação à atividade de teste; tais como, por exemplo, o ineditismo, relacionado ao fato de existir somente um dado de teste no domínio de entrada, que cubra um dado elemento requerido por um critério estrutural.

Ambos algoritmos têm uma função de *fitness* orientada a cobertura de um critério em geral, gerando uma população de indivíduos para satisfazer, um critério dado sem, entretanto, recorrer a análise do código fonte durante a geração. Ambos permitem a integração com a ferramenta JaBUTi e podem ser utilizados com diferentes critérios de teste.

A maneira como o indivíduo foi implementado permite lidar com diferentes tipos da linguagem Java.

Os algoritmos foram avaliados em um conjunto de programas e os resultados mostram que o AG Ineditismo apresentou os melhores resultados, independentemente do critério escolhido. Percebe-se um melhor desempenho das estratégias com um tamanho de população

e um número de gerações maior, mas isso também eleva o custo (tempo de execução) dos AGs. Mas esse custo é aceitável e verifica-se um aumento de cobertura que pode ser obtido automaticamente o que contribui para atividade de teste, reduzindo o esforço do testador gasto na aplicação dos critérios.

7.1 Trabalhos Futuros

Dentre as atividades que devem ser realizadas como continuidade deste trabalho, destacam-se:

- Modificar e melhorar a ferramenta TDSGen/OO implementando uma função de *fitnes*s baseada nos caminhos com objetivo de cobrir um dado elemento requerido;
- Implementação de heurísticas para a identificação de elementos não executáveis, auxiliando na avaliação dos resultados e dados gerados pela ferramenta;
- Implementar mecanismo de memorização dos melhores indivíduos para diminuir o tempo de execução do AGI;
- Implementar outros métodos de seleção e outras estratégias de hibridização.
- Realização de novos experimentos, para avaliar outros parâmetros do algoritmo, tais como o tamanho da população.
- Aplicar e avaliar os algoritmos em um conjunto de programas mais complexos.

REFERÊNCIAS BIBLIOGRÁFICAS

BIEMAN, J. M.; GHOSH, S. e ALEXANDER, R. T. 2001. *A technique for mutation of Java objects*. In *16* ***th IEEE International Conference on Automated Software Engineering****,* p. 23-26, San Diego, CA EUA, Nov. IEEE Computer Society.

BINDER, R. V. 1999 ***Testing object-oriented systems: models, patterns, and tools***. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BOTTACI, L. 2001. ***A Genetic Algorithm Fitness Function for Mutation Testing.*** In: *Seminal: Software Engineering Using Metaheuristic Innovative Algorithms. IEEE International Conference on Software Engineerring,* http://www.brunel.ac.uk//cs-stmmh2/seminal2001, October.

BOYER, R. S.; ELSPAS, B. e KARL, N. L. 1975. ***Select: A Formal System for Testing and Debugging Programs by Symbolic Execution***. *ACM SigPlan Notices,* Vol. 10(6):234-245. June.

BUDD, T. A. 1981. ***Mutation Analysis: Ideas, Examples, Problems and Prospects***. *Computer Program Testing.* North-Holand Publishing Company.

BUENO, P. M. S. e JINO, M. 2001. ***Automated Test Data Generation for Program Paths Using Genetic Algorithms***. In *13th International Conference on Software Engineering and Knowledge Engineering,* pages 1014-2019. Buenos Aires, Argentina.

CHAIN, M. L. 1991. ***Poke-Tool – Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseados em Análise de Fluxo de Dados.*** Tese de mestrado, DCA/FEE/UNICAMP, 1991.

CHEN, M. H.; TSE, T. H.; CHAN, F. T. and CHEN, T. Y. 1998. In: ***Black and white: an integrated approach to class-level testing of object-oriented programs****. ACM Transactions on Software Engineering Methoodology, 7(3), 250-295, Jul.*

CHEON, Y.; KIM, M. Y e PERUANDLA, A. 2005. ***A Complete Automation of Unit Testing for Java Programs.*** Disponível em: http://cs.utep.edu/cheon/techreport/tr05-05.pdf.

CHUNG, I. S. 1998. ***Automatic Testing Generation for Mutation Testing Using Genetic Operators.*** In: *International Conference on Software Engineering and Knowledge Engineering,* San Francisco, June.

CLARKE, L. 1976. ***A System to Generate Test Data an Symbolically Execute Programs.*** *IEEE Transactions on Software Engineering,* Vol. SE-2(3):215-222, September.

COLANZI, T. E. 1999. **Uma abordagem integrada de desenvolvimento e teste de software baseado na UML**. Orientador: Paulo Cezar Maziero. (Dissertação). USP, São Paulo SP.

CONCILIO, R. 2000. **Contribuições à solução de problemas de Escalonamento pela Aplicação Conjunta de Computação Evolutiva e Otimização de Restrições**. Orientador: Fernando José Von Zuben. (Dissertação). UNICAMP, São Paulo SP.

COSTA, E. O.; BRUNA, M. D. 2002. **Resolução de *TimeTabling* Utilizando Evolução Cooperativa**. Orientadora: Profª. Drª. Aurora Trinidad Ramirez Pozo. (Trabalho de Conclusão de Curso). UFPR, Curitiba/PR.

DELAMARO, M.E; MALDONADO, J. C. 1996. ***Proteum – A Tool for the Assessment of Test Adequacy for C Programs.*** Manual do Usuário.

De MILLO, R. A; LIPTON R. J.; SYWARD, F. G. 1978. ***Hints on Teste Data Selection for the practicing programmer.*** *Computer*.

De MILLO, R. A. e OFFUTT, A. J. 1991. *Constraint-based automatic test data generation.* ***IEEE Transactions on Software Engineering,*** *Vol. SE-17(9): 900 –910, May.*

DURAN, J. W. e NTAFOS, S. C. 1984. ***An Evaluation of Random Testing***. *IEEE Transactions on Software Engineering,* Vol. SE-10(4):438 – 444, July.

FERREIRA, L. P. 2003. **TDSGEN – Uma Ferramenta de Geração de Dados de Teste Baseada em Algoritmos Genéticos.** (Dissertação de mestrado). UFPR. Curitiba/Paraná.

FERREIRA, L. P. e VERGILIO, S. R. 2005. ***TDSGEN: An Environment Based on Hybrid Genetic Algorithms for Generation of Test Data***. In *17th International Conference on Software Engineering and Knowledge Engineering.* Taipei, Taiwan, July.

FRANKL F.G. 1987. ***The use of Data Flow Information for the Selection and Evaluation of Software Test Data***. PhD Thesis, Department of Computer Science, New York University. New York, U.S.A, Outubro.

GOEL, A. L. 1985. ***Software Reability Models: Assumptions, Limitations, and Applicability.*** *IEEE Transactions on Software Engineering*, v. 11, n. 12, Dez.

GOLDBERG, D. E. 1989. ***Genetic Algorithms in Search, Optimization and Machine Learning***. Addison-Wesley.

HARROLD, M. J.; ROTHERMEL, G. 1994. ***Performing data flow testing on classes***. *In: Second ACM SIGSOFT Symposium on Foundations of Software Engineering,* New York: ACM Press, 1994, p. 154-163.

HOFFMAN, D. e STROOPER, P. 1993. ***A case study in class testing.*** In: *CASCO 93,* IBM. Toronto Laboratory, 1993, p. 472 – 482.

HOLLAND, J. 1975. ***Adaptation in natural and artificial systems***. Ann Arbor: University of Michigan Press.

HOWDEN, W. E. 1987. ***Functional Program Testing and Analysis***. McGrall-Hill, New York.

JONES, B. F.; STHAMER, H. H. e EYRES. D. E. 1996. ***Automatic Structural Testing Using Genetic Algorithms.*** *The Software Engineering Journal,* 11:299-306.

JORGENSEN, P. 2001. ***Software Testing: A Craftman's Approach***. CRC Press, Inc., Boca Raton, FL, USA, 2001.

KOREL, B. 1990. ***Automated Software Test Data Generation.*** *IEEE Transactions on Software Engineering,* Vol.SE-16(8): 870-879. *August* .

LIASKOS, K.; ROPER, M. e WOOD, M. 2007. *Investigating Data-Flow Coverage of Classes Using Evolutionary Algorithms.* ***Genetic And Evolutionary Computation Conference. GECCO - 2007.*** pp. 1140 – 1140.

LIU, X., B. WANG, e LIU, H. 2005. ***Evolutionary search in the context of object-oriented programs***. MIC2005: *The Sixth Metaheuristics International Conference*, Vienna, Austria.

McGRAW, G. e MICHAEL, C. 1997. ***Automatic Generation of Test-Cases for Software Testing.*** Technical Report RST Corporation.

McGREGOR, J. D. e SYKES, D. A. 2001. ***A practical guide to testing object oriented software***. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

MA, Y., OFFUTT, J. e KWON, Y. R. 2005. *MuJava: An Automated Class Mutation System.* **Software Testing, Verification & Reliability**, v. 15, pp. 97 – 133.

MALDONADO, J. C. 1991. **Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.** Orientador: Prof. Dr. Mário Jino (Tese) ICMC/USP. São Carlos/SP, 1991.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F. ; SOUZA, S. R. S; DELAMARO, M. E. 1998. **Aspectos Teóricos e Empíricos de Teste de Cobertura de *Software*.** (Notas Didáticas do ICMCS - USP).

MICHAEL, C.; McGRAW, C. e SCHATZ, M. A. 2001. ***Generating Software Test Data by Evolution.*** *IEEE Transactions on Software Engineering,* Vol.SE-27(12): 1085-1110. *December*.

MYERS, G. J. 1979. ***Art of Software Testing***. John Wiley & Sons, Inc., New York, NY, USA.

NAVAUX, P. O. A. e SHUBEITA, F. M. 2003. **Computação Evolutiva e Lógica Fuzzy.** (Notas Didáticas do PPGC – UFRGS).

OFFUT, A. J. e IRVINE, A. 1995. ***Testing object-oriented software using the category-partition method.*** In: *XVII International Conference on Technology of Object-Oriented Languages Systems,* p. 293 – 304, Santa Barbara, CA, EUA, Agosto. Prentice-Hall.

PALAZZO, L. A. M. e CASTILHO, J. M. V. 1997. **Algoritmos para Computação Evolutiva.** Pelotas: Universidade Católica de Pelotas, 1997. (Relatório Técnico do Grupo de Pesquisa em Inteligência Artificial).

PARGAS, R. P.; HARROLD, M. J. e PECK, R. R. 1999. ***Test-Data Generation Using Genetic Algorithm.*** *The journal of Software Testing, Verification and Reliability,* 9:263-282.

POLO, M.; PIATTINI, M. e GARCIA-RODRÍGUEZ, I. 2009. *Decreasing the cost of mutation testing with second-order mutants.* **Software Testing, Verification & Reliability**, v. 19, pp. 111 – 131, 2009.

PRESSMAN, R. S. 1997. ***Software Engineering – a practitioner's approach*.** McGraw-Hill, 4ª edição.

PRESSAMAN, R. S. 2006. **Engenharia de Software**. McGraw-Hill, 6.

RAMAMOORTHY, C. V.; SIU-BUN, F. H. e CHEN, W. T. 1976. ***On Automated Generation of Program Test Data***. *IEEE Transactions on Software Engineering,* Vol.SE-2(4):293-300. December.

RAPPS, S., e WEYUKER, E. 1985. ***Selecting software test data using data flow infor***mation. *IEEE Transactions on Software Engineering 11*, 04 (April), 367–375.

RIBEIRO, J. C.; VEGA, F. F. e ZENHA-RELA, M . 2007. ***Using Dynamic Analysis of Java Bytecode for Evolutionary Object-Oriented Unit Testing***. Em *25th Brazilian Symposium on Computer Networks and Distributed Systems*, Belém/PA.

RIBEIRO, J. C. 2008. *Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming.*, Em ***Proc. of the GECCO '08, pp. 1819-1822***, Atlanta, Georgia, USA, July

ROPER, M.; MACLEAN, I.; BROOKS, A.; MILLER, J.; WOOD, M. 1995. ***Genetic Algorithms; Automatic Generation of Test Data***. *Technical Report RR/95/195. University of Strathelyde. Glasgow UK.*

SAGARNA, R.; ARCURI, A.; YAO, X. 2007. *Estimation of Distribution Algorithms for Testing Object Oriented Software. In****: IEEE Congress on Evolutionary Computation (CEC, 2007)****.*

SEESING, A. e GROSS, H. 2006. ***A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software***. In: *1st International Workshop on Evaluation of Novel Approaches to Software Engineering*, Erfurt, Germany, September 19—20.

TRACEY, N.; CLARK J.; MANDER, K. 1998. *Automated Program Flaw Finding Using Simulated Annealing.* ***Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*** (ISSTA'98). Florida USA, p.73-81.

TONELLA, P. 2004. ***Evolutionary Testing of Classes***. ISSTA '04: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software testing and analysis*. ACM Press: 119-128. Boston, Massachusetts, USA.

VERGILIO, S.R.; MALDONADO, J.C; JINO, M. 2007.**Geração de Dados de Teste**. In: DELAMARO, M.E.; MALDONADO, J.C.; JINO, M. *Introdução ao Teste de Software*. Ed. Campus-SBC.

VINCENZI, A. M. R.; DELAMARO, Marcio E.; MALDONADO, J. C. 2003. ***JaBUTi – Java Bytecode Undestanding and Testing***. Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (Manual do usuário), Março.

VINCENZI, A. M. R; DELAMARO, M. E.; MALDONADO, J. C. e WONG, W. E. 2006. *Establishing Structural Testing Criteria for Java Bytecode.* ***Software Parctice and Experience, 36(14): 1.512 – 1.541.*** Nov. 2006.

VINCENZI, A. M. R.; DOMINGUES, A. L. S.; DELAMARO, M. E. 2007. Teste Orientado a Objetos e de Componentes. In: Delamaro, M.E.; Maldonado, J.C.; Jino, M. *Introdução ao Teste de Software*. Ed. Campus-SBC, 2007. pp. 139, 140,

WAPPLER, S. e LAMMERMANN, F. 2005. ***Using evolutionary algorithms for the unit testing of object-oriented software***, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, Washington DC, USA.

WAPPLER, S. e LAMMERMANN, F. 2006. *Evolutionary Unit Testing of Object-Oriented Software Using a Hybrid Evolutionary Algorithm.* Em: ***Evolutionary Computation, 2006. CEC 2006. IEEE Congress on Volume , Issue , Page(s):851 – 858.***

WAPPLER, S. e WEGENER, J. 2006. **Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming.** Em: GECCO '06', Seattle/ Washignton.

WEICHSELBAUM, R. 1998. ***Software Test Automation by Means of Genetic Algorithms***. *Proceedings of the Sixth International Conference on Software Testing Analysis and Review.* Munich, Germany.

WEGENER, J.; BARESEL, A.; STHAMER, H. 2001. *Evolutionary test environment for automatic structural testing*. ***Information and Software Technology***, vol 43, p. 841-854.

WHITE, L. J. e COHEN, E. I. 1980. *A domain strategy for computer program testing.* ***IEEE Transactions on Software Engineering,*** *Vol. SE-6(3):247 – 257, May.*

APÊNDICE A
RELATÓRIOS GERADOS PELA FERRAMENTA JABUTI

A Figura A.1 ilustra o código fonte do programa TriTyp. Este programa consiste em avaliar uma sequência de três números inteiros e sua funcionalidade corresponde à classificação da entrada, se forma um triângulo equilátero, isóscele, escaleno ou não formam um triângulo.

Após a escolha do programa é necessário que o testador escolha a classe base e o caminho para localizar o resto das classes para que a ferramenta JaBUTi calcule e exiba o conjunto de classes necessário para executar o programa.

A Tabela A. 1, que representa os elementos requeridos do método type() com o critério todos-nós. O nome de cada nó é dado de acordo com o número da primeira linha do *bytecode* pertencente ao nó em questão, por exemplo, o nó 151 corresponde comando aload_0 do *bytecode* ou as linhas 44 e 45 do código normal. Isso pode ser visualizado no grafo def-uso gerado através da interface gráfica da ferramenta.

```
/*01*/ import java.io.*;
/*02*/ import java.util.*;
/*03*/ import java.io.Serializable;
/*04*/
/*05*/ public class TriTyp implements
       Serializable
/*06*/ {
/*07*/   public int i, j, k;
/*08*/   public int trityp;
/*09*/   public static int SCALENE = 1,
                    ISOSCELES = 2,
                    EQUILATERAL = 3,
                    NOT_A_TRIANGLE = 4;
/*10*/
/*11*/   public void setI(int v) {
/*12*/         i=v;
/*13*/   }
/*14*/
/*15*/   public void setJ(int v) {
/*16*/         j=v;
/*17*/   }
/*18*/
/*19*/   public void setK(int v) {
/*20*/        k=v;
/*21*/   }
/*22*/
/*23*/   / *@return
/*24*/               1 if scalene;
/*25*/               2 if isosceles;
/*26*/               3 if equilateral;
/*27*/               4 if not a triangle
/*28*/   */
/*29*/   public void type() {
/*30*/               if (i==j) { trityp=trityp+1; }
/*31*/               if (i==k) { trityp=trityp+2; }
/*32*/               if (j==k) { trityp=trityp+3; }
/*33*/
/*34*/               if (i<=0 || j<=0 || k<=0) {
/*35*/                  trityp=4;
/*36*/                  return;
/*37*/               }
/*38*/               if (trityp==0)
/*39*/               {
/*40*/                  if (i+j<=k ||
                          j+k<=i ||
                          i+k<=j) {
/*41*/                     trityp=4;
/*42*/                     return;
/*43*/               } else {
/*44*/                  trityp=1;
/*45*/                  return;
/*46*/               }
/*47*/       }
/*48*/       if (trityp>3) {
/*49*/          trityp=3;
/*50*/       } else if (trityp==1 && i+j>k)
/*51*/       {
/*52*/          trityp=2;
/*53*/       } else if (trityp==2 && i+k>j)
/*54*/       {
/*55*/          trityp=2;
/*56*/       } else if (trityp==3 && j+k>i)
/*57*/       {
/*58*/          trityp=2;
/*59*/       } else {
/*60*/          trityp=4;
/*61*/       }
/*62*/   }
/*63*/
/*64*/   public boolean equals(Object o)
/*65*/   {
/*66*/    if (!(o instanceof TriTyp)) return false;
/*67*/      TriTyp t=(TriTyp) o;
/*68*/      return (i==t.i && j==t.j
                    && k==t.k
                     && trityp==t.trityp);
/*69*/   }
/*70*/
/*71*/     public String toString()
/*72*/    {
/*73*/       String s= i + "-" + j + "-" + k + ": "
                      + trityp;
/*74*/      return s;
/*75*/    }
/*76*/  }
```

Figura A.1: Exemplo adaptado de (VINCENZI et al., 2003).

Tabela A. 1: Elementos requeridos gerados pela JaBUTi método type().

Covered	Active	Infeasible	Testing Requirement
false	true	false	151
false	true	false	181
false	true	false	245
false	true	false	97
false	true	false	213
false	true	false	274
false	true	false	63
false	true	false	32
false	true	false	197
false	true	false	165
false	true	false	90
false	true	false	229
false	true	false	77
false	true	false	11
false	true	false	42
false	true	false	145
false	true	false	70
false	true	false	113
false	true	false	173
false	true	false	269
false	true	false	237
false	true	false	205
false	true	false	129
false	true	false	53
false	true	false	84
false	true	false	0
false	true	false	157
false	true	false	21
false	true	false	261

Tabela A. 2: Dados de teste para o programa TriTyp

caso de teste 1: 1, 3, 5
caso de teste 2: 5, 5, 5
caso de teste 3: 2, 3, 2,
caso de teste 4: 3, 6, 7
caso de teste 5: 4, 10 ,8
caso de teste 6: 7, 7, 7
caso de teste 7: 4, 1, 4
caso de teste 8: 2, 5, 6
caso de teste 9: 2, 2, 1
caso de teste 10: 8, 6, 7

Executando os dez dados de teste apresentados na Tabela A. 2 chega-se a seguinte cobertura (Tabela A. 3):

Tabela A. 3: Cobertura dos elementos gerados pela JaBUTi.

Testing Criterion	Coverage	Percentage
All-Nodes-ei	29 of 45	64%
All-Nodes-ed	0 of 0	0%
All-Edges-ei	29 of 55	52%
All-Edges-ed	0 of 0	0%
All-Uses-ei	75 of 169	44%
All-Uses-ed	0 of 0	0%
All-Pot-Uses-ei	65 of 208	31%
All-Pot-Uses-ed	0 of 0	0%

No relatório gerado pela JaBUTi também são fornecidos os elementos que não foram cobertos, a porcentagem de cobertura de cada caso de teste como pode ser visto abaixo.

Caso de teste 1:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	14 of 45	31.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	9 of 55	16.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	21 of 169	12.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	10 of 208	4.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 2:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	18 of 45	40.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	13 of 55	23.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	26 of 169	15.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	25 of 208	12.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 3:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	19 of 45	42.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	14 of 55	25.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	26 of 208	12.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 4:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	16 of 45	35.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	11 of 55	20.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	12 of 208	5.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 5:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	16 of 45	35.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	11 of 55	20.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	12 of 208	5.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 6:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	18 of 45	40.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	13 of 55	23.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	26 of 169	15.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	25 of 208	12.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 7:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	19 of 45	42.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	14 of 55	25.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	26 of 208	12.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 8:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	16 of 45	35.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	11 of 55	20.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	12 of 208	5.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 9:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	18 of 45	40.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	13 of 55	23.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	28 of 169	16.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	25 of 208	12.0
All-Pot-Uses-ed	0 of 0	0.0

Caso de teste 10:

Criterion	Number Of Covered Requirements	Percentage
All-Nodes-ei	16 of 45	35.0
All-Nodes-ed	0 of 0	0.0
All-Edges-ei	11 of 55	20.0
All-Edges-ed	0 of 0	0.0
All-Uses-ei	29 of 169	17.0
All-Uses-ed	0 of 0	0.0
All-Pot-Uses-ei	12 of 208	5.0
All-Pot-Uses-ed	0 of 0	0.0

APÊNDICE B
COMANDOS DA JABUTI UTILIZADOS

Para realizar a análise de cobertura (através da linha de comando) é necessário primeiramente criar o projeto (projeto.jbt) através da classe base por meio do comando:

```
java -cp Jabuti-bin.zip: br.jabuti.cmdtool.CreateProject -b <classe> -p <projeto.jbt>
```

ou utilizando a opção -i que permite que seja fornecido o nome de um arquivo texto contendo o nome das classes a serem instrumentadas:

```
java -cp Jabuti-bin.zip: br.jabuti.cmdtool.CreateProject -b <classe> -p <projeto.jbt> -i classes.txt
```

Em seguida deve-se escolher quais classes se deseja instrumentar e o nome do projeto a ser criado. Para instrumentar utiliza-se o seguinte comando:

```
java -cp Jabuti-bin.zip br.jabuti.probe.ProberInstrum -o <arquivo.jar> -p <projeto.jbt> <classe a executar>
```

Uma vez instrumentada a classe, para executar um caso de teste utiliza-se:

```
java -cp <arquivo .jar> <classe a executar> [parâmetros para execução]
```

A importação de dados de testes pode ser feita através do *ProberLoader* (*ClassLoader* da JaBUTi). Primeiro é necessário executar o programa em teste, utilizando o *ProberLoader*, para coletar as informações de execução.

```
java -cp Jabuti-bin.zip br.jabuti.probe.ProberLoader -p <projeto.jbt> <classe>
```

Esse aplicativo chamado *ProberLoader* recebe de parâmetro o arquivo de projeto (projeto.jbt) e o nome da classe que se deseja executar. Desse modo, durante a execução dessa classe, a ferramenta é capaz de identificar quais partes do código estão sendo executadas. Toda vez que o aplicativo encerra sua execução, um novo caso de teste é gravado no arquivo.trc.

Com isso, será criado um arquivo chamado arquivo.trc que corresponde a um arquivo de trace que é utilizado pela JaBUTi para avaliar a cobertura dos critérios de testes definidos por ela.

Para avaliar o andamento da atividade de teste, relatórios de teste com diferentes níveis de granularidade (por projeto, por classe, por método, por caso de teste) podem ser gerados a fim de auxiliar o testador a decidir quando parar os teste ou quais partes ainda não foram testadas. Se a opção -o não for usada, sempre será gerado o relatório no arquivo jabuti_report.xml.

Relatório do projeto

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p <projeto.jbt> -pr -o projeto.xml
```

Relatório de classes (inclui o relatório do projeto)

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p <projeto.jbt> -cl -o classes.xml
```

Relatório de métodos (inclui o relatório do projeto e de classes)

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p <projeto.jbt> -me -o metodos.xml
```

Relatório de conjunto de teste (idem ao de projeto em termos de cobertura)

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p <projeto.jbt> -ts -o test-set.xml
```

Relatório de casos de teste (inclui conjunto de teste)

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p <projeto.jbt> -tc -o test-case.xml
```

Relatório completo

```
java -cp Jabuti-bin.zip br.jabuti.cmdtool.JabutiReport -p < projeto.jbt> -all -o completo.xml
```

APÊNDICE C
RESULTADOS PARA CADA PROGRAMA

Neste apêndice são apresentados os resultados para cada programa e cada estratégia.

Tabela C.1: Critério Todos-nós-ei para 50 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	55%	51%	60%
	100	64%	62%	64%
	200	63%	66%	75%
BubCorrecto	50	34%	54%	62%
	100	52%	48%	81%
	200	50%	51%	81%
Mid	50	81%	81%	95%
	100	83%	81%	95%
	200	86%	81%	95%
Bisect	50	38%	55%	64%
	100	32%	38%	72%
	200	39%	38%	83%

Tabela C.2: Critério Todos-nós-ei para 100 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	54%	44%	62%
	100	55%	46%	67%
	200	53%	51%	78%
BubCorrecto	50	48%	44%	66%
	100	52%	57%	83%
	200	54%	48%	84%
Mid	50	81%	81%	95%
	100	71%	57%	95%
	200	81%	48%	95%
Bisect	50	48%	55%	66%
	100	44%	38%	77%
	200	45%	55%	83%

Tabela C.3: Critério Todos-nós-ei para 200 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	60%	40%	64%
	100	59%	86%	67%
	200	52%	46%	78%
BubCorrecto	50	47%	48%	66%
	100	47%	42%	81%
	200	52%	51%	81%
Mid	50	81%	81%	95%
	100	83%	81%	95%
	200	86%	81%	95%
Bisect	50	48%	27%	83%
	100	56%	61%	83%
	200	57%	61%	83%

Tabela C.4: Critério Todos-ramos-ei para 50 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	49%	38%	49%
	100	52%	45%	56%
	200	53%	50%	67%
BubCorrecto	50	18%	40%	42%
	100	36%	31%	75%
	200	36%	37%	75%
Mid	50	67%	58%	88%
	100	69%	58%	88%
	200	69%	58%	88%
Bisect	50	35%	46%	50%
	100	30%	30%	53%
	200	36%	30%	69%

Tabela C.5: Critério Todos-ramos-ei para 100 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	50%	35%	50%
	100	44%	29%	58%
	200	58%	38%	70%
BubCorrecto	50	44%	25%	48%
	100	36%	46%	77%
	200	48%	31%	77%
Mid	50	66%	57%	88%
	100	63%	46%	88%
	200	74%	31%	88%
Bisect	50	45%	46%	52%
	100	30%	30%	54%
	200	36%	46%	69%

Tabela C.6: Critério Todos-ramos-ei para 200 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	48%	21%	50%
	100	51%	76%	52%
	200	55%	29%	67%
BubCorrecto	50	32%	31%	40%
	100	34%	28%	40%
	200	37%	37%	42%
Mid	50	67%	58%	88%
	100	69%	58%	88%
	200	69%	58%	88%
Bisect	50	40%	19%	69%
	100	40%	50%	69%
	200	46%	50%	69%

Tabela C.7: Critério Todos-usos-ei 50 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	35%	31%	34%
	100	39%	40%	40%
	200	51%	40%	52%
BubCorrecto	50	9%	26%	37%
	100	26%	18%	60%
	200	26%	26%	75%
Mid	50	71%	57%	96%
	100	73%	57%	96%
	200	74%	57%	96%
Bisect	50	39%	54%	54%
	100	33%	40%	60%
	200	40%	40%	76%

Tabela C.8: Critério Todos-usos-ei para 100 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	40%	21%	43%
	100	43%	25%	45%
	200	50%	31%	52%
BubCorrecto	50	35%	21%	44%
	100	26%	39%	70%
	200	37%	18%	76%
Mid	50	71%	57%	96%
	100	62%	39%	96%
	200	72%	18%	96%
Bisect	50	39%	54%	56%
	100	33%	40%	66%
	200	54%	54%	76%

Tabela C.9: Critério Todos-usos-ei para 200 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	21%	19%	48%
	100	30%	58%	51%
	200	35%	25%	52%
BubCorrecto	50	20%	18%	26%
	100	26%	13%	67%
	200	27%	26%	69%
Mid	50	71%	57%	96%
	100	73%	57%	96%
	200	74%	57%	96%
Bisect	50	40%	19%	48%
	100	39%	50%	55%
	200	48%	50%	76%

Tabela C.10: Critério Todos-pot-usos-ei para 50 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	20%	21%	22%
	100	28%	27%	30%
	200	33%	29%	35%
BubCorrecto	50	8%	26%	26%
	100	22%	18%	67%
	200	22%	24%	69%
Mid	50	75%	65%	96%
	100	82%	65%	96%
	200	18%	65%	96%
Bisect	50	42%	44%	48%
	100	35%	45%	50%
	200	43%	45%	73%

Tabela C.11: Critério Todos-pot-usos-ei para 100 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	20%	16%	23%
	100	23%	19%	25%
	200	27%	21%	28%
BubCorrecto	50	27%	16%	27%
	100	22%	35%	65%
	200	34%	18%	66%
Mid	50	75%	65%	96%
	100	61%	35%	96%
	200	48%	18%	96%
Bisect	50	42%	58%	50%
	100	35%	45%	50%
	200	43%	58%	50%

Tabela C.12: Critério Todos-pot-usos-ei para 200 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	21%	12%	22%
	100	25%	38%	30%
	200	34%	19%	35%
BubCorrecto	50	9%	18%	16%
	100	22%	16%	30%
	200	23%	24%	35%
Mid	50	75%	65%	96%
	100	77%	65%	96%
	200	63%	65%	96%
Bisect	50	42%	20%	50%
	100	28%	50%	50%
	200	45%	50%	50%

A média do tempo gasto (tempo de execução da ferramenta) com cada estratégia está nas Tabelas 6.14 a 6.16.

Tabela C.13: Média do tempo de execução para 50 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	0:01:02	01:12:15	0:44:56
	100	0:01:03	00:40:15	0:49:02
	200	0:01:01	00:57:14	1:03:08
BubCorrecto	50	0:01:08	0:52:26	0:44:20
	100	0:01:16	0:56:23	0:46:38
	200	0:01:10	0:50:57	0:43:45
Mid	50	0:01:07	0:53:40	0:44:31
	100	0:01:10	0:55:37	0:46:46
	200	0:01:12	0:51:23	0:46:01
Bisect	50	0:01:06	0:53:15	0:44:02
	100	0:01:11	0:55:22	0:47:36
	200	0:01:09	0:51:37	0:42:56

Tabela C.14: Média do tempo de execução para 100 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	0:01:00	02:01:16	1:25:50
	100	0:01:04	02:06:13	1:38:34
	200	0:01:02	01:57:56	2:06:19
BubCorrecto	50	0:01:05	01:45:35	1:24:35
	100	0:01:09	01:50:48	1:32:32
	200	0:01:15	01:41:21	2:03:45
Mid	50	0:01:07	01:46:51	1:28:28
	100	0:01:09	01:51:00	1:33:33
	200	0:01:08	01:42:48	2:03:45
Bisect	50	0:01:06	01:46:16	1:28:28
	100	0:01:08	01:50:51	1:32:08
	200	0:01:07	01:43:06	1:29:28

Tabela C.15: Média do tempo de execução para 200 gerações.

Programa	Tamanho População	Geração Aleatória	Algoritmo Genético	Algoritmo Genético Modificado
TriTyp	50	0:01:32	04:08:58	3:24:21
	100	0:01:23	04:10:08	3:30:06
	200	0:01:15	03:52:08	4:37:26
BubCorrecto	50	0:01:05	03:32:32	2:36:47
	100	0:01:09	03:41:18	3:06:06
	200	0:01:20	03:24:36	3:12:22
Mid	50	0:01:06	03:33:03	2:55:47
	100	0:01:06	03:39:48	2:57:57
	200	0:01:07	03:23:15	3:07:57
Bisect	50	0:01:06	03:33:36	2:55:05
	100	0:01:08	03:40:28	3:04:04
	200	0:01:08	03:24:57	2:59:50

Tabela C.16: Média de cobertura para o programa TriTyp.

	Gerações	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	61%	51%	42%	27%
	100	54%	51%	44%	23%
	200	57%	51%	29%	29%
AG	50	60%	44%	37%	26%
	100	47%	34%	26%	19%
	200	57%	42%	34%	23%
AG Ineditismo	50	66%	57%	42%	29%
	100	59%	59%	47%	25%
	200	70%	56%	50%	29%

Tabela C.17: Média de cobertura para o programa BubCorrecto.

	Gerações	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	45%	51%	20%	17%
	100	51%	43%	33%	28%
	200	49%	34%	24%	11%
AG	50	51%	36%	23%	23%
	100	50%	34%	26%	23%
	200	47%	32%	18%	19%
AG Ineditismo	50	75%	57%	57%	54%
	100	78%	67%	63%	53%
	200	76%	41%	54%	27%

Tabela C.18: Média de cobertura para o programa Mid.

	Gerações	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	83%	68%	73%	58%
	100	78%	68%	68%	61%
	200	83%	68%	73%	72%
AG	50	81%	58%	57%	65%
	100	62%	45%	38%	39%
	200	81%	58%	57%	65%
AG Ineditismo	50	95%	88%	96%	96%
	100	95%	88%	96%	96%
	200	95%	88%	96%	96%

Tabela C. 19: Média de cobertura para o programa Bisect.

	Gerações	Todos-nos-ei	Todos-ramos-ei	Todos-usos-ei	Todos-potenciais-usos-ei
Geração Aleatória	50	36%	34%	37%	40%
	100	46%	37%	42%	40%
	200	54%	40%	42%	38%
AG	50	44%	35%	67%	45%
	100	49%	41%	49%	54%
	200	50%	40%	39%	40%
AG Ineditismo	50	73%	57%	63%	57%
	100	75%	58%	66%	50%
	200	83%	69%	60%	50%

As Tabela C.16 a C.19 apresentam a média de cobertura para os programas TriTyp, BubCorrecto, Mid e Bissect considerando cinco execuções, tamanho da população igual a duzentos, pois foi a que obteve melhor cobertura dos critérios, e três números de gerações: 50, 100 e 200. Observou-se que não houve muita diferença nos valores obtidos para cada critério analisado de acordo com cada estratégia, com o valor de desvio padrão entre 1% e 3%.