Uma Abordagem Evolutiva para a Geração Automática de Dados de Teste

Bruno Teixeira de Abreu

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Abordagem Evolutiva para a Geração Automática de Dados de Teste

Bruno Teixeira de Abreu

Agosto de 2006

Banca Examinadora:

- Prof^a. Dr^a. Eliane Martins
  Instituto de Computação – UNICAMP (Orientadora)
- Prof^a. Dr^a. Silvia Regina Vergilio
  Departamento de Informática – UFPR
- Prof. Dr. Ricardo Anido
  Instituto de Computação – UNICAMP
- Prof. Dr. Orlando Lee (Suplente)
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Abreu, Bruno Teixeira de
Ab86a Uma abordagem evolutiva para a geração automática de dados de teste / Bruno Teixeira de Abreu -- Campinas, [S.P. :s.n.], 2006.

Orientador : Eliane Martins; Fabiano Luís de Sousa
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software – Testes. 2. Computação evolutiva. 3. Teste orientado a caminhos. 4. Otimização extrema generalizada. 5. Algoritmos genéticos. I. Martins, Eliane. II. Sousa, Fabiano Luís de. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Titulo. (mjmr/imecc)

Título em inglês: An evolutionary approach for automatic test data generation.

Palavras-chave em inglês (Keywords): 1. Software testing. 2. Evolutionary computation. 3. Path testing. 4. Generalized extremal optimization. 5. Genetic algorithms.

Área de concentração: Engenharia de software

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Eliane Martins (IC-UNICAMP)
Profa. Dra. Silvia Regina Vergilio (DI-UFPR)
Prof. Dr. Ricardo Anido (IC-UNICAMP)

Data da defesa: 25/08/2006

Programa de Pós-Graduação: Mestrado em Ciência da Computação

Termo de Aprovação

Dissertação defendida e aprovada em 25 de agosto de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Silvia Regina Vergilio
Departamento de Informática / UFPR

Prof. Dr. Ricardo de Oliveira Anido
IC/UNICAMP

Profa. Dra. Eliane Martins
IC/UNICAMP

Prof. Dr. Fabiano Luis de Sousa
Instituto Nacional de Pesquisas Espaciais

Para meus adoráveis pais,
Carmélio e Cely.

Resumo

O teste é uma atividade importante do processo de desenvolvimento de software, e automatizar a geração de dados de teste contribui para a redução dos esforços de tempo e recursos. Recentemente foi mostrado que os algoritmos evolutivos, tal como os Algoritmos Genéticos (AGs), são ferramentas valiosas para a geração de dados. Este trabalho avalia pela primeira vez o desempenho de um algoritmo evolutivo proposto recentemente, a Otimização Extrema Generalizada (em inglês, *Generalized Extremal Optimization*, GEO), na geração de dados de teste para cobrir um subconjunto de caminhos de um programa, com ou sem *loops*. Sete programas muito conhecidos e utilizados como *benchmarks* por outros autores foram escolhidos como estudos de caso, e o desempenho do GEO foi comparado com o de um AG e o *Random-Test* (RT). Uma aplicação real do Instituto Nacional de Pesquisas Espaciais (INPE) também foi testada para validar a pesquisa, e as comparações de desempenho incluíram uma variação do AG utilizado nos *benchmarks*. Para os *benchmarks* e a aplicação real, o uso do GEO exigiu muito menos esforço computacional para gerar os dados do que os AGs, e a cobertura média de caminhos obtida por ele foi muito semelhante à dos AGs. Além disso, o GEO também exigiu muito menos esforço computacional no ajuste interno de parâmetros do que os AGs. Estes resultados indicam que o GEO é uma opção muito atraente a ser utilizada na geração de dados de teste.

Abstract

Software testing is an important activity of the software development process and automating test data generation contributes to reduce cost and time efforts. It has recently been shown that evolutionary algorithms, such as the Genetic Algorithms (GAs), are valuable tools for test data generation. This work assesses for the first time the performance of a recently proposed evolutionary algorithm, the Generalized Extremal Optimization (GEO), on generating test data to cover a subset of paths of a program, with or without loops. Seven well known benchmark programs were used as study cases, and the performance of GEO was compared to the one of a GA and Random-Test (RT). A real application from Instituto Nacional de Pesquisas Espaciais (INPE) was also tested in order to validate the research, and the performance comparisons included one variation of the GA used in the *benchmarks*. For the benchmark programs and the real application, using GEO required much less computational effort to generate test data than using the GAs, and GEO's average coverage was very similar to GA's. Besides this, it also required much less computational effort on internal parameter setting than the GAs. These results indicate that GEO is a very attractive option to be used for test data generation.

Agradecimentos

Primeiramente, agradeço a Deus por ter me inundado de saúde, alegria e muita paz no coração durante todo este tempo. Não basta estar feliz somente por fora, também é importante estar feliz por dentro, e Ele me mostrou este caminho.

Obrigado a meus pais, Carmélio e Cely, e a meu irmão, Gleison, por serem tão confidentes e maravilhosos, e também por estarem sempre do meu lado, me incentivando a crescer e a vencer, me apoiando nas batalhas e me confortando nas derrotas. A grande distância geográfica que nos separa é insignificante perto do amor e confiança que temos um pelo outro. Amo vocês! Agradeço também a toda minha família por todos os momentos maravilhosos que vivemos neste tempo, principalmente aos meus avós por serem um exemplo de vida admirável.

Não teria colhido os frutos deste trabalho sem o apoio incondicional e a orientação de duas pessoas fabulosas: Eliane Martins e Fabiano Sousa, meus orientadores. É impressionante como uma ótima orientação transforma uma pessoa, e adquiri um nível de maturidade em pesquisa impressionante graças a estas duas pessoas. O incentivo de vocês dois me fizeram superar barreiras e encarar de frente, sem medo nem receios, o desconhecido. Este trabalho é uma conquista nossa! Obrigado por todo o incentivo, por todas as oportunidades oferecidas e pelas inúmeras portas que vocês me ajudaram a abrir, tanto profissionais quanto acadêmicas. Agradeço também ao Prof. Dahab pelo apoio e por ter me apresentado a Profa. Eliane nos primeiros meses do curso.

Não me esqueço também dos professores da UFMG que me incentivaram a enfrentar o desafio de vir para a Unicamp: Roberto e Mariza Bigonha, Rodolfo Resende, Osvaldo Farhat e Jeroen van de Graaf. Agradeço em especial ao professor José de Siqueira, outro orientador fabuloso com quem tive o privilégio de trabalhar durante a graduação.

A alegria de concluir mais uma etapa de meus estudos também é fruto do incentivo e da amizade de todos aqueles com quem convivi durante estes anos. São inúmeras pessoas, e provavelmente corro um sério risco de me esquecer de alguém. Mesmo assim, faço questão de agradecer aos meus amigos Goiás, Goiano, Cebola, Rocha, Augustão, Davizão, Terezinha, Dennys Filosofia, Dennis No Notion, Daniel Cohen, Korea, Bruninha, VP, Vitim e Márcio Baiano pela convivência extremamente sadia em nosso *flat*. Obrigado também aos meus outros amigos e amigas de Campinas, principalmente à Fer, Vanessa e

Dineide pelos inúmeros bate-papos.

Aos amigos que fiz no Instituto Nacional de Pesquisas Espaciais (INPE), em especial à Ana Maria, Fátima Matiello, Valdivino Santiago Jr., Ana Sílvia, Wendell, Danilo e Anderson. Muito obrigado a todos vocês pelo auxílio técnico e também pelas oportunidades oferecidas. Obrigado à turma engraçadíssima com quem convivi no LSD: meus amigos Moronte, Leonel, Leo, Patrick, Bittencourt, Neumar, Ivan, Tomita, Camila e Cláudião. Um agradecimento especial a todos os membros do grupo de pesquisa de V&V do IC. Obrigado também à turma do *trekking* (Barthô, Borin e Juliana) pela diversão garantida nos finais de semana, mesmo porque quase sempre nossa equipe se perdia nas trilhas quando eu era o navegador. Agradeço também a outros amigos do IC: LeoBSD, Dani-Linux, Vignatti, Sheila, Rafael Mamão, André Atanásio, Carlos Eduardo, Celso e Carlão. Viva aos nossos churrascos!!! Viva ao Serviço de Informação da Carne (SIC) também!!! Viva ao StarClean, ao Ponto Final e aos outros barzinhos de Campinas que tornaram esta caminhada mais agradável!!! Viva a dança de salão, fonte de amizade e descontração!!!

Muita motivação também veio de meus amigos de Minas Gerais. Agradeço a todos da minha turma de graduação da UFMG, que é um exemplo a ser seguido por ter nascido unida e continuar assim até hoje. Devo um agradecimento especial aos meus grandes amigos de BH: Sasá, Guigui, Xará, Marcinha, Augustão, Brenim, Tiôla, o casal Polly e Adriano, Aguimar Jr., João Ayala, Kátia Lage, Eder, Leandro, Luiz Barbosa, Dolabela, Radicchi, Vinícius Coelho, Larissa Petroianu, Mari Coelho, as irmãs Angélica e Marina Lima, Diego Nogueira, Robert Pereira, Stenim, Isa e Gisele Cardoso. Pessoal, as inúmeras "baladas"em Belo Horizonte foram muito divertidas. Obrigado também pelas oportunidades oferecidas! Os bate-papos também foram extremamente valiosos para mim! Cada um de vocês tem uma infinidade de qualidades admiráveis que fazem parte de mim hoje.

Não posso deixar de agradecer de coração aos meus grandes amigos que me hospedaram em suas residências em Belo Horizonte: Filipe Melo, Pequitito, Talles, Aguimar Jr., Eder, Leandro, Ronaldinho, Bernardo, e meus primos Xará, Guigui e Sasá. Apesar da vida extremamente agitada em BH, o que me deixava fora de suas casas praticamente o dia inteiro, os momentos em que parávamos para conversar foram muito bons. Obrigado pela confiança, pela hospedagem e pela amizade! Agradeço também aos "Quatro Amigos"(Fred, Tiôla, Biju e Felipe) e à Glau pela amizade construída no 1o. grau em Caratinga-MG e que perdura até hoje, apesar da distância entre todos nós.

Finalmente, agradeço a esta universidade que dispensa comentários, a Unicamp, e também ao apoio financeiro que recebi durante todo este tempo. Foram várias as fontes: o "paitrocínio", o INPE através dos projetos PLAVIS e QSEE, o CNPq e o Instituto de Computação da Unicamp. Obrigado a todos vocês por tornarem possível a realização deste meu sonho, o título de Mestre em Ciência da Computação!

"O algoritmo da vida: seguir a vontade,
impraticável em sua simplicidade."
– Guilherme Abreu Faria

"Viver é a coisa mais rara do mundo.
A maioria das pessoas apenas existe."
– Oscar Wilde

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Acrônimos

AE Algoritmo Evolutivo

AG Algoritmo Genético

AG-CD Algoritmo Genético com *Crossover* de Dois pontos

CDG *Control Dependence Graph*

CFG *Control Flow Graph*

CLP *Constraint Logic Programming*

EE Estratégia Evolutiva

EO *Extremal Optimization*

GD *Gradient Descent*

GEO *Generalized Extremal Optimization*

HD *Hamming Distance*

ICFG *Interprocedural Control Flow Graph*

INPE Instituto Nacional de Pesquisas Espaciais

IPP *Inverse Path Probability*

MAPSAR *Multi-Application Purpose Synthetic Aperture Radar*

NEHD *Normalized Extended Hamming Distance*

OBDH *On-Board Data Handling*

ORCAS Observação de Raios Cósmicos Anômalos e Solares na Magnetosfera

PE Programação Evolutiva

PG Programação Genética

RMI *Remote Method Invocation*

RT *Random-Test*

SA *Simulated Annealing*

SGA *Simple Genetic Algorithm*

SOC *Self-Organized Criticality*

SP *Subject Program*

SUT *Software Under Test*

TS *Tabu Search*

Capítulo 1

Introdução

Teste é uma atividade importante do processo de desenvolvimento de software, que tem como objetivo revelar falhas (em inglês, *fault*[1]) em um software. Mesmo sendo uma atividade cara em termos de esforço e custos, ela provê a última palavra a partir da qual a qualidade pode ser auferida e as falhas reveladas [Pre97], apesar de não garantir a ausência delas [DDH72]. Quando o software em construção cresce em termos de tamanho e complexidade, o projeto, implementação e execução de testes manuais tornam-se tarefas caras, difíceis e pouco produtivas.

Além disso, cada pedaço de software desenvolvido por qualquer empresa do mundo precisa ser testado para garantir um mínimo de qualidade ao cliente. Há vários exemplos famosos de problemas ou até mesmo desastres que ocorreram devido a testes insuficientes antes da entrega do software [Wik06b]. Um aspecto muito importante na atividade de testes é avaliar o quanto um conjunto de dados de teste exercitaram o software. Neste contexto, foram definidos vários critérios de teste para auxiliar esta avaliação. Uma vez que um ou mais critérios são escolhidos, é necessário definir como criar (ou selecionar) dados de teste que atendam a um ou mais critérios selecionados. Por outro lado, a enumeração exaustiva das entradas de um software é inviável para qualquer software de tamanho razoável. A geração aleatória de entradas também não é adequada por não satisfazer requisitos bem específicos durante a execução do software.

Tendo em vista esta dificuldade, este trabalho de mestrado trata do problema da geração automática de dados de teste para o teste orientado a caminhos, utilizando uma abordagem evolutiva para gerar os dados. A automação contribui, dentre outras coisas, para melhorar a qualidade do software que é entregue ao cliente, visto que a execução de um número maior de testes ajuda a aumentar a confiança no funcionamento do software produzido; outra contribuição é a redução de custos e esforços de tempo consumidos na atividade de testes pelas empresas [Pre97], apesar de que na indústria o processo de testes

[1] Terminologia utilizada na área de Tolerância a Falhas.

é geralmente manual [McM04]. Ould [Oul91] sugere que a automação desta tarefa é o aspecto mais importante para automatizar os testes. De acordo com McMinn [McM04], a automação da geração de dados de teste tem sido limitada, o que justifica o interesse crescente de muitos pesquisadores no uso de métodos de otimização para a geração automática dos dados.

No entanto, a automação da geração de dados de teste possui limitações. Estas geralmente estão associadas ao tamanho e à complexidade do software em teste (em inglês, *Software Under Test*, SUT), além do fato de que, em geral, a geração de dados de teste é um problema indecidível para muitos critérios de teste [MMS01]. Por exemplo, o critério de caminhos visa selecionar todos os caminhos possíveis do SUT; no entanto, sabe-se que isto é impossível devido à existência de caminhos que não são executáveis, além dos *loops*, que podem levar a um número infinito de caminhos. Outro ponto importante é que a geração aleatória de dados não é confiável, uma vez que estes têm poucas chances de exercitar comportamentos muito específicos do software [MMS01].

Uma solução promissora para contornar estas limitações é o uso de metaheurísticas[2] de busca, que são. Estas metaheurísticas são métodos de otimização, e o uso destes métodos implica na conversão do problema de geração de dados de teste para um problema de otimização. Na matemática, otimização é a disciplina cujo objetivo é encontrar os máximos e mínimos de funções, possivelmente sujeitos a restrições. Um problema de otimização matemática, também conhecido como problema de otimização, pode ser escrito da seguinte forma [SR02, Van98]:

$$\begin{cases} minimizar & f_0(x), & \\ sujeito\ a & f_i(x) \leq 0 & i = 1, ..., m \\ & g_p(x) = 0 & p = 1, ..., n \\ & x_j^{min} \leq x_j \leq x_j^{max} & j = 1, ..., k \end{cases} \tag{1.1}$$

onde $x = (x_1, ...x_k)$ é o vetor das variáveis de projeto (com k componentes) e a funcão f_0 é a função objetivo. As funções f_i, $i = 1, ..., m$, e g_p, $p = 1, ..., n$, são as funções de restrição de desigualdade e igualdade, respectivamente. A última função de restrição é a lateral, que define o intervalo de variação (de um valor mínimo x^{min} até um máximo x^{max}) para cada componente do vetor de variáveis de projeto. Um vetor x^* é ótimo se ele possui o menor valor de função objetivo dentre todos os vetores que satisfaçam todas as restrições impostas. Em outras palavras, se x^* é ótimo, então ele é uma solução para a Equação 1.1. Observe que uma função a ser otimizada pode ter um ou mais máximos ou mínimos locais e globais ($L_{\{max,min\}}, G_{\{max,min\}}$), como mostra a Figura 1.1.

No caso da automação da geração de dados de teste, o problema consiste em converter

[2]Uma heurística que combina outras heurísticas de uma maneira eficiente, no intuito de resolver uma classe genérica de problemas computacionais [Wik06d].

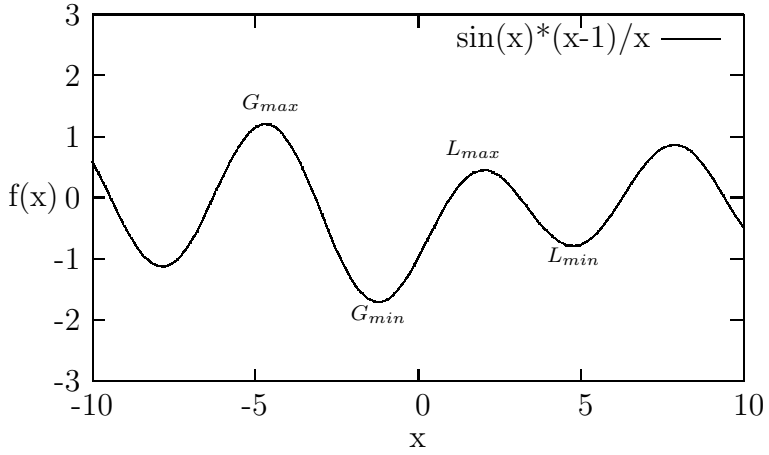

Figura 1.1: Exemplo de uma função objetivo em um espaço de projeto.

os critérios de teste para funções-objetivo. Estas, por sua vez, comparam soluções e as avaliam em relação ao objetivo da busca (por exemplo, gerar um dado de teste que execute a instrução $a = b$ de um software). Em suma, a função objetivo conduz a busca pelos dados de teste para as regiões mais promissoras do espaço de busca, aumentando as chances de gerar um dado que atenda ao critério definido.

Uma das metaheurísticas mais utilizadas para a geração automática de dados de teste é o Algoritmo Genético (AG) [WH06, MMS01, PHP99, Sth96, PGGZ94, SGJ92], inicialmente proposto por Holland [Hol75]. O AG é uma metaheurística baseada nos princípios da teoria da evolução natural de Darwin que realiza uma busca global por soluções viáveis no espaço de projeto do problema que está sendo tratado. Ele pertence a uma categoria mais geral conhecida como Algoritmos Evolutivos (AEs), e tem sido utilizado por muitos pesquisadores de uma maneira eficaz para gerar dados para satisfazer a diversos critérios de teste de um SUT [MS04, MMS01, PHP99, Sth96].

Recentemente um novo AE, a Otimização Extrema Generalizada (em inglês, *Generalized Extremal Optimization*, GEO) [Sou02], foi proposto. O GEO foi desenvolvido originalmente como uma melhoria do método da Otimização Extrema (em inglês, *Extremal Optimization*, EO) [BP01b] que, por sua vez, inspirou-se no modelo evolutivo de Bak-Sneppen [BS93]. O GEO tem sido aplicado com sucesso em problemas reais de projeto ótimo [GSRM04a, SVR04, SRPG03] e foi competitivo com outros métodos estocásticos[3] em funções-teste [SRPG03]. Sua vantagem principal em comparação com os outros algoritmos estocásticos é que ele possui um único parâmetro a ser ajustado (os outros algoritmos possuem no mínimo três), facilitando seu processo de configuração para obter o melhor desempenho em uma dada aplicação.

[3]Métodos estocásticos possuem em sua solução um grau de aleatoriedade.

O GEO nunca havia sido utilizado em aplicações na Engenharia de Software até a proposta deste trabalho de mestrado, e seus resultados de sucesso em outros problemas de otimização, tanto práticos quanto teóricos, motivaram duas perguntas: (1) o GEO obtém resultados bons e competitivos para a geração de dados de teste, tal como nos outros problemas onde ele já foi aplicado?; (2) O desempenho do GEO supera o de um AG, um dos algoritmos mais utilizados para a geração de dados de teste?. Estas perguntas compõem a motivação para avaliar o GEO, um algoritmo potencialmente competitivo com outras metaheurísticas de busca, pela primeira vez, em um problema da Engenharia de Software.

Este trabalho apresenta respostas para as duas perguntas do parágrafo anterior, e avalia a aplicabilidade do GEO em testes de software, especificamente na geração automática de dados para teste orientado a caminhos (em inglês, *Path Testing*). O teste orientado a caminhos prevê a geração de dados para cobrir pelo menos uma vez um subconjunto de caminhos pré-selecionados de um SUT, incluindo caminhos com *loops*. A grande maioria dos programas possuem *loops* (e.g. uma construção *for* ou *while*) e estes devem ser exercitados um número mínimo de vezes [Bin99, Pre97, Bei95], mesmo sabendo que a existência deles dentro do software pode levar a um número infinito de caminhos [Bin99, Pre97]. O trabalho de pesquisa também mostra a importância do processo de ajuste dos parâmetros dos algoritmos para a melhoria de seus desempenhos.

Para atingir estes objetivos, sete programas bem conhecidos e utilizados pela comunidade científica como *benchmarks* foram selecionados para os experimentos, sendo que cinco destes possuem caminhos com *loops*. Para validar a proposta do trabalho, um estudo de caso foi feito utilizando uma aplicação real do Instituto Nacional de Pesquisas Espaciais (INPE). O GEO foi comparado com a geração aleatória de dados de teste, chamada de *Random-Test* (RT), e com o algoritmo genético que foi a base para todas as variações subseqüentes, o Algoritmo Genético Simples (em inglês, *Simple Genetic Algorithm*, SGA). Para compará-los, foram utilizados três critérios: a cobertura média de caminhos, o número de execuções do programa em testes e o tempo consumido até o final do processo de geração de dados. No estudo de caso da aplicação real, o GEO também foi comparado com uma variação do algoritmo genético, diferente do SGA.

O próximo capítulo apresenta conceitos básicos de testes de software, incluindo considerações a respeito da automação de testes. O Capítulo 3 introduz os Algoritmos Evolutivos, apresentando brevemente os conceitos envolvidos e as limitações do uso destes algoritmos. Este mesmo capítulo apresenta com detalhes o algoritmo genético utilizado neste trabalho para as comparações de desempenho com o GEO. Na seqüência, o Capítulo 4 dedica-se a uma descrição breve dos trabalhos relacionados de outros autores. O GEO é apresentado com detalhes no Capítulo 5, enquanto que o Capítulo 6 descreve a abordagem para geração automática de dados de teste utilizada neste trabalho. Os Capítulos 7 e 8

apresentam e discutem os resultados utilizando os sete estudos de caso simples, e o estudo de caso da aplicação real, respectivamente. As conclusões e sugestões para trabalhos futuros encontram-se no Capítulo 9.

Capítulo 2

Os testes de software

O teste é uma atividade cujo objetivo é revelar falhas em um SUT. Ela é um elemento crítico para a garantia de qualidade e representa a revisão final da especificação, projeto e codificação [Pre97]. Esta atividade é um dos elementos que fazem parte de um tópico mais abrangente, chamado de Verificação & Validação (V&V). A verificação diz respeito ao conjunto de atividades que garantem que o software implemente uma determinada função corretamente. Já a validação envolve outras atividades cujo objetivo é garantir que o software atende às necessidades especificadas pelo usuário.

Apesar da V&V englobar um domínio grande de atividades, a melhor atividade pela qual a qualidade pode ser aferida e as falhas encontradas ainda é a de testes [Bin99, Pre97], apesar de ela demandar um grande esforço em termos de tempo e custos [Bur03]. Idealmente os testes garantem a ausência de falhas no software mas, na realidade, eles somente revelam a presença de falhas, e não garantem a ausência delas [Mye79].

Os testes devem evitar que requisitos do software em construção deixem de ser atendidos, pois problemas com requisitos são os mais graves do ponto de vista do usuário [Som03, Pre97]. Além disso, os testes devem ser planejados antes da fase de implementação do software, mesmo porque os documentos necessários para seu desenvolvimento normalmente estão disponíveis na fase de análise, imediatamente anterior à de implementação. Em relação ao progresso dos testes, estes devem começar visando módulos individuais e, após isso, partir para blocos maiores como, por exemplo, vários módulos integrados.

Um módulo pode ser testado utilizando todas as combinações de entrada possíveis, exercitando, dessa forma, todas as estruturas possíveis do software. Isto implica em testes exaustivos, que não são recomendados em virtude do esforço computacional e tempo demandados [Bin99, Pre97]. Uma outra forma interessante de se testar o mesmo módulo é selecionar entradas de maneira aleatória com a esperança de revelar falhas críticas. No entanto, a criação e seleção de testes que tenham um potencial maior para revelar falhas exige um entendimento das funcionalidades do software, domínio de entrada e saída, e do

ambiente de uso do código em testes.

Em suma, um teste bom e eficiente é aquele que é planejado e tem uma probabilidade grande de revelar a presença de uma ou mais falhas. Um teste não pode ser redundante devido à limitação de tempo e de recursos, bem como não deve avaliar uma parte muito grande do SUT, ou muito pequena.

2.1 Técnicas de teste

As duas técnicas principais para a construção de teste são a caixa-branca e caixa-preta.

2.1.1 Testes caixa-branca

A técnica de testes caixa-branca, também conhecida como técnica de testes estruturais [Som03, Bin99], tem como foco avaliar a estrutura interna do software [Som03, Pre97]. Sendo assim, é indispensável o conhecimento da estrutura interna dele, o que faz com que o código ou uma representação do software em pseudo-código esteja disponível. Os critérios pertencentes a esta técnica são classificados com base no fluxo de controle do programa, fluxo de dados e na complexidade [dMW01].

Um recurso muito utilizado para representar a estrutura interna do software é o grafo de fluxo de controle (em inglês, *Control Flow Graph*, CFG). Um CFG representa a estrutura estática de um programa ou método, como mostra a Figura 2.1. Em sua representação, regiões contíguas de código que não determinam desvios de fluxo (decisões ou instruções *goto*), conhecidas como blocos básicos, são representados como nós em um grafo e as arestas entre nós indicam o fluxo possível do programa. Uma decisão é um operador ou função que retorna um valor booleano (verdadeiro ou falso). Um ciclo em um CFG pode implicar na existência de um *loop* no código e, na maioria das representações, existem dois blocos bem definidos: o bloco de entrada, demarcando o ponto de início do programa, e o bloco de saída, por onde o fluxo de controle do programa se encerra.

2.1.2 Tipos de análise

A análise de fluxo de dados tem como foco as interações entre as definições de variáveis, também conhecidas como $defs$, e suas referências no programa, também conhecidas como *usos*. Uma definição de variável equivale a uma operação de inicialização, declaração ou atribuição onde o estado interno da variável é alterado. Os *usos* são referências a variáveis previamente definidas, e podem ser divididos em *c-usos* e *p-usos*, dependendo de como é feita a referência à variável no programa. No caso de um *c-uso*, a variável é referenciada em uma instrução de computação, enquanto que no *p-uso*, ela é referenciada em uma

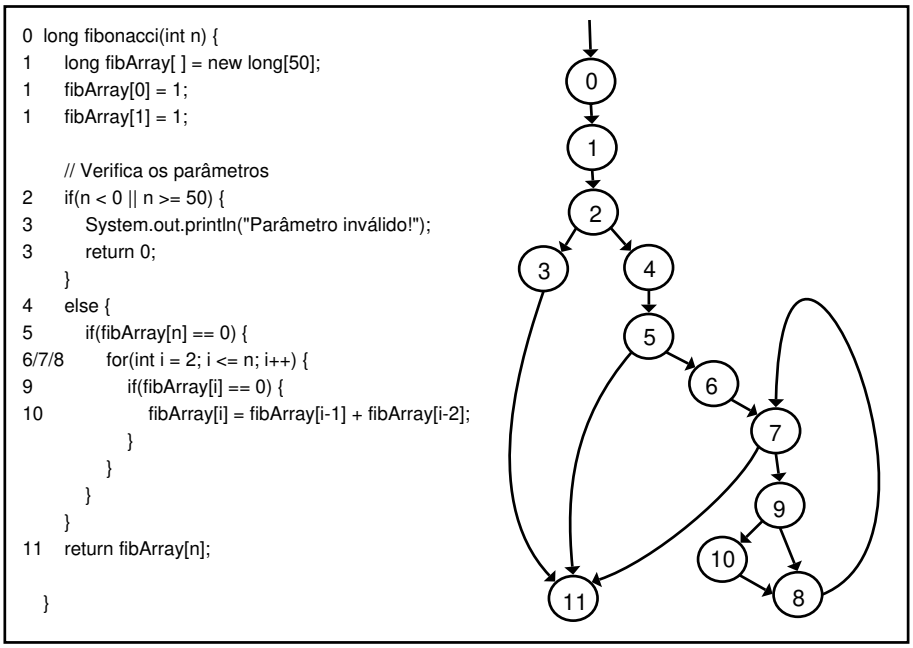

Figura 2.1: Código-fonte de um programa e seu respectivo CFG.

decisão do programa. O propósito deste tipo de análise é determinar as definições de todas as variáveis do programa e os usos que podem ser afetados por estas definições.

A Figura 2.2 mostra o código-fonte de um programa e seu grafo de fluxo de controle. Os números na primeira coluna da esquerda indicam as instruções, e os da segunda coluna indicam os blocos básicos do programa. A partir deste código-fonte, é possível extrair um conjunto de *c-usos* e *p-usos* para cada variável. Por exemplo, a variável m foi definida no bloco básico 1, mas não foi utilizada em nenhuma parte do programa. Já a variável x foi definida no bloco 1 e usada nas decisões dos blocos 2, 5, 9 e 12; o uso neste caso ocorreu na transição do bloco 2 para 3, caso o resultado da comparação fosse verdadeiro, e de 2 para 5, caso fosse falso. A variável x também foi usada nos blocos 6 e 13 para computações. Sendo assim, os pares (α, β) de def-(*c-uso*) para x, onde α e β são blocos básicos do CFG, seriam (1,6) e (1,13). No caso dos pares (θ, λ) de def-(*p-uso*), onde θ é um bloco básico e λ uma aresta entre dois blocos básicos, este seriam (1,2-3), (1,2-5), (1,5-6), (1,5-8), (1,9-10), (1,9-12), (1,12-13) e (1,12-15). Note que as $defs$ e os *c-usos* são associados aos nós do CFG, enquanto os *p-usos* são associados às arestas entre os blocos básicos.

Já a análise de fluxo de controle tem como foco a ordem na qual as instruções do programa são executadas. Uma instrução de fluxo de controle é uma instrução que ao ser executada pode provocar uma mudança no fluxo de controle do programa, ou seja, a execução deixa de seguir o fluxo normal e sequencial das instruções. Os tipos de instruções

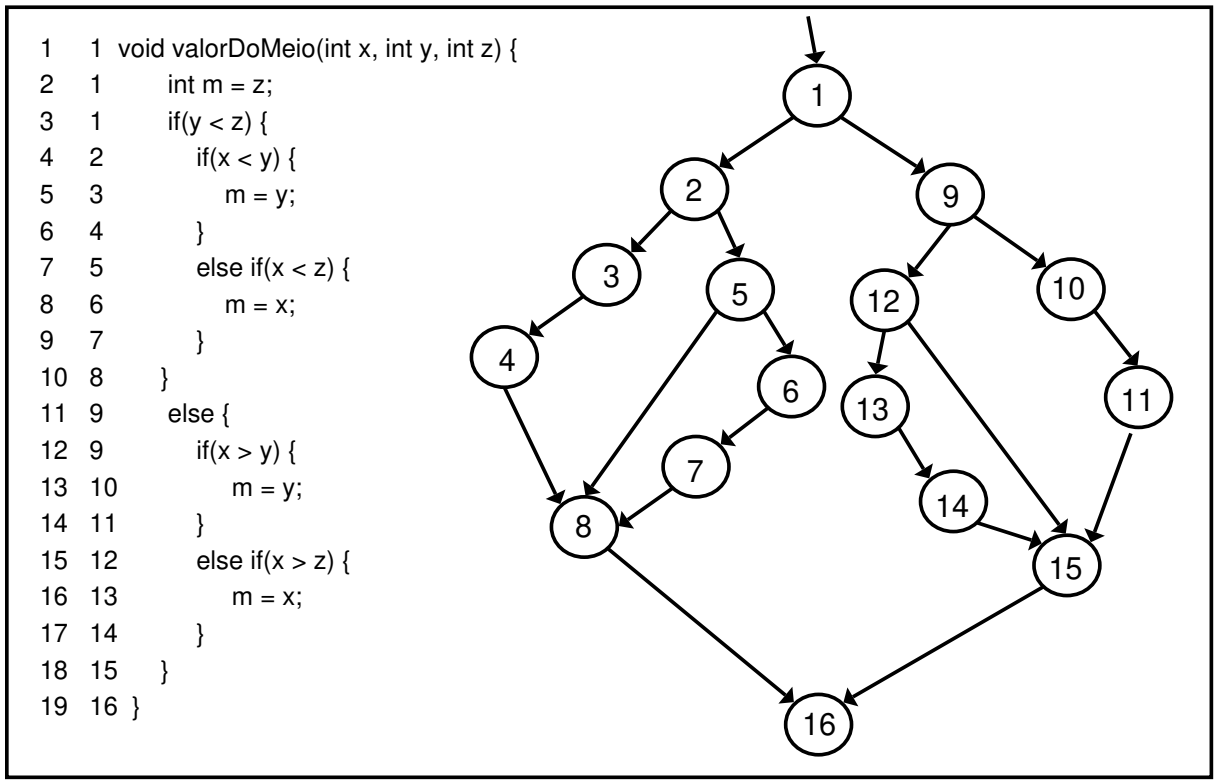

Figura 2.2: Código-fonte, número das instruções, blocos básicos e CFG de um programa.

de fluxo de controle variam de acordo com as linguagens de programação, mas podem ser categorizados em [Wik06c]: instruções de salto, escolha, repetição, subrotinas, e de parada. Por exemplo, os blocos básicos 2, 5 e 9 da Figura 2.2 (representados pelos nós 2, 5 e 9) são alguns exemplos de instruções de escolha. No caso deste tipo de análise avalia-se o fluxo de controle do programa, e os vários fluxos do programa são representados através de caminhos ao longo do CFG. Um caminho é uma seqüência finita de nós conectados por arestas, logo os blocos básicos (2,5,8) formam um caminho. Já um caminho completo deve obrigatoriamente começar no bloco básico inicial e terminar no final; neste caso, um dos caminhos completos seria formado pelos blocos básicos (1,2,5,8,16).

A seleção dos testes é feita de forma a exercitar elementos internos estruturais específicos, com a intenção de saber se eles estão funcionando apropriadamente. Por exemplo, geralmente os testes são criados com o objetivo de exercitar todas as instruções ou decisões (avaliando a condição tanto para verdadeiro quanto para falso) de um módulo ou método do SUT. Em virtude da grande quantidade de tempo consumida no projeto, execução e análise dos resultados quando se utiliza esta técnica, ela é normalmente aplicada somente a partes pequenas de um software. Os critérios englobados por esta técnica são bastante úteis para revelar os seguintes tipos de falhas [Bur03]: projeto e codificação

de controle, lógica e seqüência, inicialização e fluxo de dados. A seção a seguir descreve alguns destes critérios e entra em detalhes no critério de caminhos, cujos conceitos aplicam-se ao teste orientado a caminhos, utilizado neste trabalho de mestrado.

Critérios

Há uma série de critérios de teste associados à técnica caixa-branca [Bur03], sendo que o de instruções (ou todos os nós), de decisões (ou todos os arcos) e o de caminhos (ou todos os caminhos) só são alguns exemplos [dMW01]. Cada critério define um conjunto de requisitos do SUT que devem ser satisfeitos, e ele pode ser utilizado para verificar se um conjunto de testes já foi executado suficientemente [Bur03, dMW01]. A necessidade de exercitar determinados atributos e propriedades do código-fonte conduziu a uma abordagem chamada de análise de cobertura. Esta, por sua vez, é utilizada para definir os objetivos dos testes e avaliar os dados utilizados.

Dessa forma, o conjunto de testes pode ser melhorado para que atributos e propriedades do SUT que ainda não foram testadas sejam avaliadas com a inclusão de novos testes. A Figura 2.3 mostra uma hierarquia destes critérios. Quanto mais forte o critério, mais alta sua posição na hierarquia e maior a chance dos testes revelarem falhas, apesar de que o esforço de tempo e recursos para a criação e execução dos mesmos também é maior.

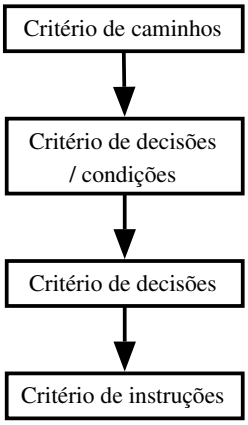

Figura 2.3: Hierarquia de alguns dos critérios de testes caixa-branca.

No critério de instruções (ou teste de instruções), todas as instruções do código-fonte do SUT devem ser exercitadas pelo menos uma vez. No caso da Figura 2.1, todos os nós do grafo deveriam ser executados pelo menos uma vez. O analista de testes pode definir um grau de cobertura, que é a percentagem de instruções que deverão ser cobertas pelos testes. Uma das razões para o grau de cobertura seja definido como um número abaixo de 100% é que algumas instruções do código-fonte podem ser inalcançáveis [Bur03].

Um critério de teste mais forte que o de instruções é o de decisões (ou ramos). Uma decisão é composta por uma ou mais condições e, no critério de decisões, cada construção de linguagem que represente um elemento de decisão no código-fonte (*if/else, switch/case, loop*) deverá ser executada uma vez com a saída retornando verdadeiro, e outra retornando falso. Voltando à Figura 2.1, todas as arestas do grafo devem ser exercitadas pelo menos uma vez. Este critério de teste contém o critério de instruções, uma vez que exercitar todos as decisões implica em exercitar todas as instruções do software. O critério de decisões/condições, por sua vez, avalia além das decisões as condições que fazem parte dele (no caso de decisões com múltiplas condições).

O critério de caminhos tem por objetivo exercitar pelo menos uma vez todos os caminhos possíveis de um SUT [Bei90]. Um caminho é uma seqüência de nós no CFG, iniciando no nó de entrada e terminando no nó de saída [Bur03]. Dentre os inúmeros caminhos do método da Figura 2.1, podemos citar: $0-1-2-3-11$, $0-1-2-4-5-11$, $0-1-2-4-5-6-7-11$, $0-1-2-4-5-6-7-9-8-7-11$, onde o '$-$' representa uma aresta entre os nós do CFG. Este critério é o mais forte dentre todos os existentes, pois ele tem a habilidade de encontrar falhas que só são encontradas caso instruções e decisões sejam executadas em uma determinada ordem. Apesar disso, cobrir todos os caminhos possíveis de um SUT é geralmente impossível e computacionalmente impraticável por razões do tipo [Bur03]:

1. O número de caminhos em um programa é exponencial ao número de decisões dele [Bin99], e a cobertura de todos os caminhos em um SUT muito complexo pode se tornar uma tarefa cansativa e praticamente impossível dependendo deste número.
2. *Loops* em um programa, um tipo de construção de linguagem muito comum em programas, pode conduzir a um número infinito de caminhos, tornando a cobertura de todos os caminhos computacionalmente impossível. É relevante comentar que caminhos com um número diferente de iterações de um mesmo *loop* são caminhos distintos.
3. Há caminhos no programa que nunca serão executados, independentemente do dado de teste. Estes caminhos também são conhecidos como não-executáveis, e sua existência está associada à lógica inerente ao programa.

Em virtude disto, o critério de caminhos normalmente envolve a seleção de um subconjunto de caminhos do SUT a serem cobertos [LY01]. O problema do número infinito de caminhos devido à presença de *loops* pode ser evitado limitando o número de iterações de cada *loop* [Sth96].

No caso do teste orientado a caminhos, um subconjunto de caminhos do SUT é pré-

selecionado[1] e dados de teste são gerados na tentativa de cobrir, pelo menos uma vez, cada caminho do subconjunto.

2.1.3 Testes caixa-preta

Na técnica de testes caixa-preta, o analista de testes considera o SUT como uma caixa fechada, cujo funcionamento interno é totalmente desconhecido. O único conhecimento à disposição dele é 'o que' o SUT faz, mas não o 'como'. O tamanho do SUT utilizando esta técnica pode variar de um módulo simples até um software completo. A descrição do comportamento ou funcionalidades dele pode ser obtida a partir de uma especificação formal ou um conjunto bem definido de pré-condições e pós-condições [Bur03]. Outra fonte de informações é o documento de especificação de requisitos, que normalmente descreve as funcionalidades do software, suas entradas e saídas esperadas [Pre97]. O analista aplica as entradas especificadas no SUT, executa o teste e verifica se as saídas produzidas são iguais às previstas na especificação. O uso desta técnica é bastante útil para revelar falhas no processo de construção do software, visto que o que está especificado é o que deve ser construído; sabendo que os testes são projetados a partir de uma especificação considerada correta, caso o software não se comporte de acordo com o especificado, uma ou mais falhas são reveladas. Esta técnica também é conhecida como técnica de testes funcionais ou baseados na especificação [Som03, dMW01, Bin99].

Critérios

Dentre os critérios de testes caixa-preta, esta seção irá apresentar o critério de partições em classes de equivalência e o de análise de valores-limite. Outros exemplos de critérios são o de transição de estados e o grafo de causa e efeito [dMW01].

No caso da partição em classes de equivalência, o domínio de entrada do SUT é particionado em classes de forma que todos os membros de uma mesma classe fazem com que o software se comporte de maneira equivalente. O número finito de classes de equivalência permite que o analista de testes selecione um determinado membro de uma classe como o representante dela, visto que é assumido que todos os membros dela são processados da mesma forma pelo SUT. Uma das vantagens desta técnica é que ela elimina os testes exaustivos. Além disso, ela guia o analista de forma que ele selecione um subconjunto de entradas com uma probabilidade grande de revelar uma falha; vários autores descrevem como definir as classes de equivalência [Som03, Bur03, Bin99, Pre97, Bei95]. Uma outra vantagem é que o método permite que o analista de testes cubra um domínio maior de entradas com um subconjunto pequeno de entradas selecionadas de uma

[1]Este subconjunto pode ser selecionado utilizando, ou não, uma gama de critérios de teste, inclusive o de caminhos (apesar das limitações práticas enumeradas anteriormente).

classe de equivalência. É importante comentar que devem ser consideradas tanto classes válidas quanto inválidas que, por sua vez, representam entradas erradas ou inesperadas.

O critério de análise de valores-limite enriquece a partição em classes de equivalência, uma vez que ele requer que o analista de testes selecione entradas com valores próximos às bordas de forma que as bordas superiores e inferiores de uma classe de equivalência sejam cobertas pelos testes. Testes que exploram valores de entrada nos limites (no final do intervalo ou imediatamente fora do intervalo) têm uma chance maior de revelarem falhas [Cla76], pois avaliam erros comuns na construção do software, e.g. uma condição de parada de um comando *while* ou de um *for*. Além disso, as regras para a definição dos testes são bem definidas e claras.

É importante comentar que tanto o critério de partições quanto o de análise de valores-limite não consideram combinações de condições de entrada do programa. Estes dois critérios asseguram a cobertura de todas as condições de entrada de maneira individual.

2.2 Geração automática de dados de teste

O uso de ferramentas para a automação de testes tem grande potencial de aumentar a produtividade, reduzindo tempo e riscos, além de melhorar a qualidade do software e do processo [Bur03]. Esta seção dedica-se exclusivamente à geração automática de dados de teste (somente dados de entrada). Sendo assim, não será abordada a geração automática de oráculos de teste. Um oráculo é uma fonte confiável de resultados esperados para os testes [Bin99], podendo ser desde a especificação do programa, uma tabela de exemplos ou simplesmente o conhecimento do programador a respeito de como o software deve funcionar.

Para a automação da geração dos dados, é importante considerar dois aspectos: o critério de testes e a ferramenta para a geração dos dados. Como foi visto na Seção 2.1.2, o critério avalia a qualidade dos dados gerados e fornece informações a respeito do final do processo de geração. A avaliação de qualidade está diretamente relacionada à satisfação dos requisitos de teste definidos pelo usuário. Satisfazer um requisito implica na existência de um dado que exercite uma ou mais funções ou instruções do software relacionados ao requisito. De acordo com [MMS01], há estudos que concluem que a existência dos critérios aumentam as chances de um conjunto de testes revelar falhas.

Na maioria das vezes, independente do critério utilizado, quando um dado de teste não satisfaz um requisito, este é alterado de forma que o requisito seja efetivamente satisfeito. Em virtude da dificuldade de se fazer isto manualmente, algoritmos para a geração de dados de testes são utilizados para a automação deste processo. Estes algoritmos são as ferramentas para a geração dos dados, e a geração pode ser guiada utilizando basicamente dois tipos de análise, que é a análise de fluxo de dados e a de fluxo de controle, detalhadas

na Seção 2.1.2.

2.2.1 Abordagens principais

As três abordagens principais para automatizar a geração de dados de teste são [MMS01]: aleatória, execução simbólica, e dinâmica. A menos utilizada nos dias de hoje é a abordagem de execução simbólica, visto que esta apresenta inúmeros problemas na medida em que a complexidade do software aumenta, como será visto na Seção 2.2.1.

Abordagem aleatória

Esta abordagem simplesmente seleciona aleatoriamente valores de entrada que pertencem ao domínio das variáveis e os aplica no SUT. A geração aleatória de dados de teste é interessante pelo fato de ela gerar dados que se assemelham aos utilizados operacionalmente pelos usuários [Sth96]. Além disso, é um método relativamente barato para a geração inicial de dados [Bin99], e normalmente exige mais do SUT do que testes selecionados manualmente, mesmo porque os dados gerados podem ser completamente diferentes uns dos outros.

Por outro lado, entradas aleatórias têm grande chance de nunca exercitarem ambas as condições de uma decisão que, por exemplo, verifique se dois valores são iguais. Isto torna esta abordagem ineficiente, em virtude de raramente prover a cobertura necessária ao software [MMS01, Sth96]. Por exemplo, no caso de um programa complexo, talvez um dado tenha que satisfazer requisitos de teste muito específicos. Sendo assim, o número de entradas que satisfaçam a um requisito pode ser muito pequeno perto do número total. Isto implica em uma probabilidade muito pequena de selecionar aleatoriamente um dado de entrada que satisfaça ao requisito. Myers [Mye79] chega a afirmar que esta é a pior abordagem para automação de testes. Apesar disso, esta abordagem é útil quando o SUT é simples e não envolve restrições muito complexas [MMS01].

Abordagem de execução simbólica

Na década de 70 a grande maioria dos geradores automáticos de dados de testes utilizavam a abordagem de execução simbólica, proposta por King [Kin76] em 1976. Ela consiste em atribuir valores simbólicos às variáveis do software de forma a obter uma caracterização matemática e abstrata do que o software faz. Ou seja, obtém-se uma representação funcional dos caminhos do software, nomes simbólicos são atribuídos às variáveis de entrada, e os caminhos são avaliados a partir da interpretação das instruções e dos caminhos, de acordo com estes nomes simbólicos [Sth96].

A geração das expressões simbólicas é um passo necessário, e estas expressões representam os requisitos necessários para executar um determinado caminho ou decisão [Cla76]. A execução simbólica traz como resultado um conjunto de restrições de igualdade e desigualdade com relação às variáveis de entrada, sendo que estas restrições podem ser lineares ou não-lineares. Estas restrições, por sua vez, definem um subconjunto do domínio das variáveis que levarão à execução de um caminho escolhido. A expressão simbólica para uma variável carrega informação suficiente tal que, se valores numéricos forem atribuídos às entradas, um valor numérico pode ser obtido para a variável.

Porém, a derivação sistemática e manipulação das expressões simbólicas é computacionalmente caro, ainda mais se realizada para um número elevado de caminhos [MMS01, Sth96]. Outra desvantagem é que a abordagem apresenta problemas para software com variáveis que dependem da execução de *loops*, variáveis dependentes de vetores, referências a ponteiros e chamadas de métodos [MMS01, Kor90]. Tecnicamente, qualquer função computacional pode ser calculada sem o uso de vetores e ponteiros, mas a realidade mostra que os sistemas de software construídos raramente deixam de utilizar estes recursos de linguagem de programação. Um software que utiliza vetores e ponteiros pode ser, em teoria, executado simbolicamente [MMS01], mas estas construções inevitavelmente complicam e muito a aplicação da abordagem para a geração de dados.

Um exemplo recente de gerador de dados de testes utilizando esta abordagem é o trabalho de Meudec [Meu01]. Nele, o autor propõe um gerador de dados de testes baseado na execução simbólica de software utilizando a linguagem de programação de restrições lógicas (em inglês, *Constraint Logic Programming*, CLP) como *framework*. Além disso, apresenta um protótipo de ferramenta, ATGen, que utiliza a abordagem e visa alcançar uma cobertura de decisões de 100% em software escrito em SPARK Ada, caso não existam decisões inviáveis. Além deste, Howden [How77], Clarke [Cla76] e Boyer et al. [BEL75] apresentam outras ferramentas para a geração automática de dados.

Abordagem dinâmica

A abordagem dinâmica para a geração de dados de testes foi introduzida em 1976 por Miller e Spooner [MS76]. Esta abordagem trata partes do software como funções que são avaliadas através da execução do software. Na medida em que os dados são gerados, estes são avaliados e melhorados utilizando métodos de otimização de funções, que guiam a busca por dados adequados. Em suma, a idéia nesta abordagem é que se um requisito de teste desejado não é satisfeito, os dados coletados durante a execução dos testes são úteis para determinar quais foram os dados que se aproximaram de satisfazer o requisito. Dessa forma, os melhores podem ser alterados até que um deles satisfaça o requisito desejado [MMS01] como, por exemplo, cobrir uma determinada decisão.

O método de otimização é o responsável por gerar os dados de teste, e ele utiliza

informações específicas extraídas do SUT durante sua execução. Todas as informações necessárias para o funcionamento do método de otimização podem ser extraídas do SUT através da instrumentação do software. Além disso, a instrumentação e a execução do SUT permitem saber se um dado de teste satisfez ou não um ou mais requisitos de teste que foram especificados.

Muitos trabalhos que tratam da geração automática de dados de testes fazem o uso desta abordagem [WH06, MS04, DTB03a, BJ02, MMS01, PHP99, GN97, JSE96, FK96, Sth96, Kor90], em virtude de suas vantagens frente às outras abordagens. A estratégia que é mais encontrada envolve a redução do problema de geração de dados de teste para um problema de otimização, cujos conceitos básicos foram apresentados no Capítulo 1. Como conseqüência desta redução, o problema será representado por um conjunto de restrições (quando existirem) e uma função matemática (também chamada de função objetivo) que deverá ser minimizada ou maximizada.

A proposta da abordagem dinâmica é mais genérica e resolve muitos dos problemas da abordagem com execução simbólica do software. Todavia, ela envolve o uso de heurísticas que não garantem a geração de dados de teste até mesmo quando estes existem. Além disso, muitos métodos de otimização aplicados são muito simples [MMS01], e boa parte deles apresentam problemas quando a função objetivo possui mínimos e/ou máximos locais. Uma função pode possuir vários pontos que minimizam o seu valor, mas muitos não representam o menor valor que a função pode atingir. O ponto onde a função é o menor valor possível é chamado de mínimo global, enquanto que os demais são chamados de mínimos locais, visto que a função atinge nestes pontos os menores valores quando comparado a outros em sua vizinhança. A Figura 2.4 mostra uma função que possui vários máximos e mínimos, locais e globais ($L_{\{max,min\}}, G_{\{max,min\}}$).

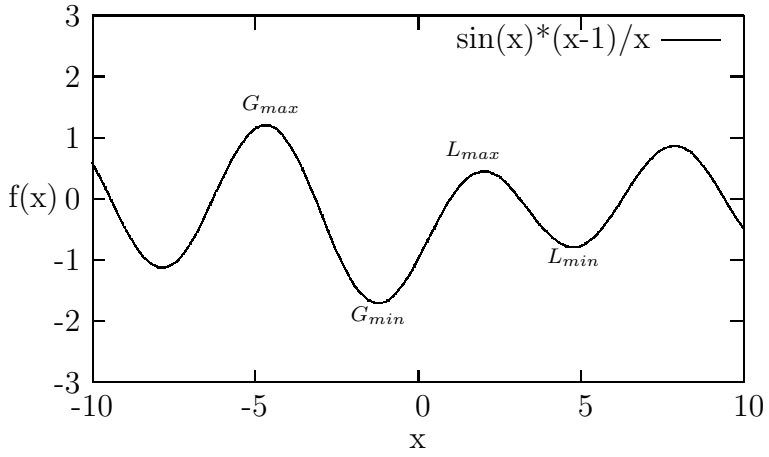

Figura 2.4: Máximos e mínimos locais e globais em um espaço de busca.

Um exemplo de método que apresenta problemas quando há mínimos/máximos locais é o método dos gradientes (em inglês, *Gradient Descent*, GD). O GD é tido como o método de otimização de funções padrão [MMS01], e trabalha fazendo mudanças sucessivas em um dado de teste de forma que cada mudança diminua o valor da função objetivo. Apesar de ser um método simples e que encontra soluções rapidamente, o método GD pode falhar caso um mínimo local seja encontrado. Isto ocorre porque o método considera somente uma pequena região do espaço de busca, implicando em um número limitado de variações de entradas. Sendo assim, todas as entradas avaliadas na vizinhança de um ponto podem resultar em um aumento no valor da função objetivo, indicando que o ponto em questão é o melhor. Porém, o ponto em questão pode se tratar simplesmente de um mínimo local, fazendo com o que a busca fique estagnada e retorne como resultado uma solução que não é a melhor. O mesmo problema ocorre quando a função possui regiões grandes no espaço de busca onde a função objetivo não altera seu valor, também chamadas de platôs.

Uma forma de reduzir estes problemas é reiniciar a busca de tempos em tempos utilizando uma solução inicial diferente, de forma a explorar melhor o espaço de busca. No entanto, é claro que para espaços de busca não-triviais, os resultados obtidos com este método são altamente dependentes da solução inicial [McM04]. Uma classe de algoritmos que realiza uma busca global no espaço de projeto, minimizando este tipo de problema, é a dos Algoritmos Evolutivos, que serão descritos em detalhes no próximo capítulo.

Capítulo 3

Os Algoritmos Evolutivos

Os Algoritmos Evolutivos (AEs) são métodos de busca estocástica baseados nos princípios e modelos da evolução natural. A idéia principal por trás destes algoritmos é evoluir uma população de indivíduos (candidatos a solução para o problema) através de competição, recombinação e mutação, de forma que a aptidão média da população (qualidade das soluções) seja sistematicamente melhorada dentro do ambiente (problema) em questão. A Tabela 3.1, retirada de [ES03], relaciona os termos normalmente empregados na evolução natural aos utilizados na descrição de um problema quando são utilizados AEs. Já a Figura 3.1 apresenta os passos básicos de qualquer AE, sendo que o processo *seleção I* é responsável pela seleção dos pais, enquanto o *seleção II* seleciona os indivíduos que sobreviverão, compondo assim a nova população do ambiente.

Tabela 3.1: Analogia entre a evolução natural e a resolução de problemas.

Evolução natural		Resolução de problemas
Ambiente	$\Leftrightarrow$	Problema
Indivíduo	$\Leftrightarrow$	Candidato a solução
Aptidão	$\Leftrightarrow$	Qualidade

Os AEs podem ser utilizados em diferentes problemas, e o processo evolutivo dos candidatos à solução é estocástico e guiado pela definição de parâmetros ajustáveis [ES03]. Sua utilização é crescente, visto que alguns métodos de otimização clássicos geralmente não apresentam um desempenho satisfatório quando submetidos a problemas reais. Além disso, já foi observado que problemas cujas características envolvam interações não-lineares, caos, variações temporais e funções que não sejam facilmente descritas, são resistentes a abordagens de otimização clássicas [ES03].

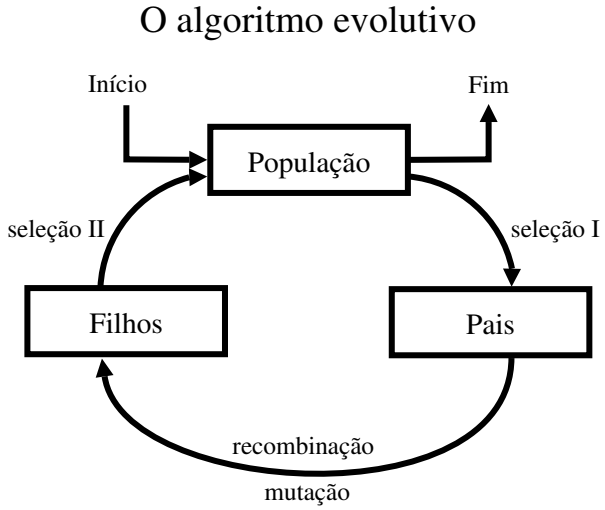

Figura 3.1: Descrição alto-nível de um AE.

3.1 Os princípios da evolução natural

O conjunto mais aceito de teorias evolutivas atualmente é o Neodarwinismo, ou Teoria Sintética da Evolução, reunindo estudos de diversos pesquisadores, entre eles, Charles Darwin, Jean-Baptiste Lamarck, Gregor Mendel e Hugo De Vries [Coe03].

A primeira explicação para a evolução das espécies foi dada por um dos primeiros biólogos contemporâneos, Jean-Baptiste Lamarck. Sua proposta ficou conhecida como Lamarckismo, e pregava basicamente duas afirmações: (1) a influência do ambiente produz mudanças físicas no indivíduo de uma espécie, e (2) esse indivíduo transmite as modificações para seus descendentes, que nascem adaptados. Lamarck morreu em 1829, e sua teoria foi parcialmente derrubada décadas depois pelos experimentos de August Weismann. Weismann comprovou empiricamente que a segunda afirmação de Lamarck não era válida através de experimentos com várias gerações de ratos de laboratórios.

Charles Darwin concordava com os princípios de Lamarck, porém percebeu que eles eram insuficientes para explicar a variedade das espécies e suas condições de vida. Em 27 de dezembro de 1831, Darwin embarcou no navio inglês *HMS Beagle* para uma viagem ao redor do mundo, com o objetivo de aprimorar sua experiência como naturalista. Todo o fundamento empírico para sua teoria foi obtido durante e após esta viagem, que durou 5 anos. Especialmente durante a visita ao arquipélago de Galápagos, Darwin percebeu que os pássaros de mesmo gênero possuíam características diferentes entre as diversas ilhas, sendo que a diferença era mais acentuada nos bicos deles. Darwin associou isto aos tipos de sementes das ilhas, ou seja, algumas variações morfológicas dos bicos adequam-se melhor a determinados tipos de sementes. Dessa forma, pássaros com bicos mais apropria-

dos para adquirir as sementes teriam mais facilidade de prosperarem do que pássaros com bicos com uma morfologia que dificultasse a aquisição delas e, conseqüentemente, acabariam se extinguindo (porque não conseguiriam se alimentar, por exemplo) [Wik06a]. Em outras palavras, indivíduos com características boas (por exemplo, um bico apropriado para um tipo de semente) seriam selecionados naturalmente e produziriam descendentes, pelo fato de estarem mais adaptados às condições do ambiente em que vivem. Esta observação, entre outras, permitiram que Darwin chegasse à famosa teoria da Evolução Natural pela Seleção Natural, também conhecida como Darwinismo. De acordo com ela, somente os indivíduos mais adaptados ao ambiente poderão sobreviver, se reproduzir e assim transmitir suas características adaptativas ao seus descendentes.

Gregor Mendel realizou um trabalho pioneiro para explicar como as características herdadas eram transmitidas entre os indivíduos. Uma vez que indivíduos de uma espécie não são idênticos entre si, é possível diferenciá-los utilizando duas características principais: as adquiridas e as herdadas. As características adquiridas ao longo da vida são evidentes como, por exemplo, a mudança na cor da pele devido à exposição ao sol. Apesar disso, as herdadas nem sempre correspondem às nossas expectativas, mesmo porque nem sempre as características dos pais se manifestam nos filhos. A pesquisa de Mendel utilizando cruzamento entre ervilhas solucionou esta dúvida, e fez com que ele propusesse as Leis de Mendel, ou Leis da Hereditariedade, que explicavam como uma característica pode se manifestar em um indivíduo mesmo que seus pais não a manifestem, apesar de gerações anteriores aos seus pais a manifestarem. Sua pesquisa somente foi reconhecida e comprovada dezesseis anos após sua morte, em 1900, por dois pesquisadores que reproduziram e publicaram suas experiências, Erich Tschermak e Hugo De Vries. De Vries, além disso, propôs o fenômeno da mutação, uma anomalia ocorrida no processo de reprodução.

A evolução dos indivíduos ocorre a partir de quatro procedimentos básicos: competição, seleção, recombinação e mutação. A competição implica na disputa entre os membros da população pelos recursos do ambiente, que por sua vez são finitos. Os indivíduos mais bem adaptados ao ambiente terão uma facilidade maior para aproveitar os recursos disponíveis e, conseqüentemente, terão uma probabilidade maior de sobreviverem. A recombinação se encarrega de transferir o código genético dos indivíduos (pais) para os seus descendentes. Por fim, a mutação é a responsável pela introdução de diversidade nas espécies [BFM97].

3.2 Os princípios evolutivos aplicados à computação

O objetivo de fazer com que computadores resolvam problemas automaticamente é algo central à inteligência artificial e ao aprendizado de máquina, sendo esta grande área englobada pelo que Turing chamou de “inteligência de máquina” [KP03].

Em 1948, Turing publicou um trabalho entitulado *Intelligent Machinery*, onde identificava três maneiras pelas quais seria possível fazer com que a inteligência de uma máquina fosse competitiva com a do homem [Tur92]. Uma das maneiras citava a existência da busca genética ou evolutiva combinando genes e utilizando como critério de avaliação um "valor de sobrevivência" [KP03]. Ele também percebeu que uma abordagem produtiva e possível para alcançar a inteligência da máquina envolveria um processo evolutivo no qual uma descrição de um programa de computador (o material hereditário) passaria por modificação progressiva (mutação), sendo guiado pela seleção natural.

Os trabalhos de Friedberg [Fri58] e Friedberg et al. [FDN59] estão entre as primeiras tentativas de simular a evolução. Estes focaram na melhora gradual de um programa de computador, auto-codificado através da manutenção de instruções associadas com maior freqüência a resultados de sucesso. O resultado de vários experimentos indicaram que era possível evoluir programas, mas o mesmo processo falhou na geração de funções mais complexas como, por exemplo, um *and*. Além disso, foi constatado que adaptar um programa desta forma consumia mais tempo do que realizar uma busca completamente aleatória [Fog90].

Três anos depois, Bremermann [Bre62] associou a evolução biológica a um processo de otimização. A busca é feita em uma superfície (que representa uma função matemática) indicando a aptidão das possíveis combinações dos genes disponíveis. As funções-objetivo utilizadas por Friedberg [FDN59, Fri58] eram muito complexas, o que fez Bremermann escolher experimentos evolucionários nos quais as superfícies de resposta envolvidas na busca eram teoricamente bem entendidas, e onde existiam técnicas numéricas para chegar à solução ótima. Ele conjecturou que a reprodução sexual trazia sucesso onde a evolução assexuada estava estagnada, mas os diversos esquemas de reprodução que ele aplicou em seus experimentos não trouxeram resultados tão bons quanto ele esperava [Fog90].

Apesar da idéia de utilizar os princípios da evolução natural para a resolução de problemas ter suas raízes no final da década de 40, foi durante a década de 60 que emergiram os primeiros algoritmos evolutivos [ES03]: as Estratégias Evolutivas (EEs) e a Programação Evolutiva (PE). Juntamente com as EEs e PE, os Algoritmos Genéticos (AGs) e a Programação Genética (PG) compõem as quatro abordagens principais dos AEs. É importante ressaltar que durante muito tempo, as abordagens que constituem hoje os AEs foram desenvolvidas independentemente uma das outras. O primeiro esforço organizado com intenção de integrar as pesquisas na área aconteceu em 1990 em Dortmund, na Alemanha, com a realização do *workshop Parallel Problem Solving from Nature* (PPSN'90) [Coe03].

3.3 Uma breve descrição dos principais AEs

Embora as quatro abordagens principais dos AEs sejam bem semelhantes em alto nível, cada uma delas implementa o AE de uma maneira diferente, e as diferenças básicas entre elas estão na estrutura que representa os indivíduos, mecanismos de seleção e formato dos operadores genéticos. A seguir, as principais abordagens são brevemente descritas, com destaque maior aos AGs pelo fato desta abordagem estar intimamente relacionada a este trabalho de pesquisa.

3.3.1 As Estratégias Evolutivas

Ao mesmo tempo em que Holland desenvolvia os AGs na década de 60 e início da década de 70, Ingo Rechenberg e Hans-Paul Schwefel trabalhavam na Alemanha desenvolvendo as EEs. Nos anos 60, a primeira EE foi desenvolvida no *Hermann Föttinger Institute for Hydrodynamics* da Universidade Técnica de Berlim, Alemanha [Coe03]. Ela resultou de experimentos em junho de 1964 em um túnel de vento, e foi chamada de EE-(1+1) pelo fato de um pai produzir somente um filho por geração.

As duas estratégias básicas de EEs são divididas de acordo com o mecanismo de seleção, e são conhecidas como (μ, λ)-EE e $(\mu + \lambda)$-EE, onde o símbolo μ indica o número de pais, e λ indica o número de filhos que serão produzidos em uma única geração. Na (μ, λ)-EE, os λ filhos substituem os μ pais, enquanto que na $(\mu + \lambda)$-EE, os λ filhos competem com os μ pais pela sobrevivência (os piores λ dentre todos os $\mu + \lambda$ indivíduos são descartados) [BS02].

A primeira estratégia normalmente mantém uma diversidade maior de indivíduos na população, mas um indivíduo muito bom tem chances de não sobreviver até o final do processo evolutivo. Já a segunda estratégia permite que uma elite de indivíduos, que são pais, domine o processo até que um indivíduo melhor seja obtido. Dessa forma, um indivíduo muito bom não é perdido e sobrevive até o final do processo evolutivo.

A representação utilizada é um vetor de tamanho fixo com valores reais, sendo que cada elemento do vetor representa uma característica do indivíduo. As EEs enfatizam o fenótipo (expressão das características comportamentais) ao invés do genótipo (estrutura do código genético) [Ang96], e o operador principal é a mutação, onde um valor aleatório é adicionado a cada elemento do vetor para produzir um novo descendente. Outro operador utilizado é a recombinação intermediária, onde é feita uma média aritmética dos elementos dos vetores de dois pais, elemento por elemento, para produzir um novo descendente.

As EEs são geralmente aplicadas a problemas de otimização com valores reais, e tendem a enfatizar mutação ao invés de *crossover*. Além disso, estes algoritmos são geralmente utilizados com tamanho de população menores (por exemplo, de 1 a 20 indivíduos) do que os AGs. Por fim, da mesma forma que os AGs, o seu desempenho depende da confi-

guração adequada dos seus parâmetros internos de controle. Porém, nas EEs estes podem ser ajustados automaticamente através de um mecanismo de auto-adaptação [Coe03], ao invés do mecanismo manual geralmente utilizado nos AGs.

3.3.2 A Programação Evolutiva

A PE tem suas origens na década de 60 graças aos trabalhos de Lawrence J. Fogel [Fog62], que teve a idéia de desenvolver inteligência artificial a partir da simulação da evolução. Em seus trabalhos, métodos evolutivos foram aplicados na tentativa de aprimorar máquinas de estado finitas.

A máquina de estado finita foi escolhida por representar bem o problema evolutivo elaborado por Fogel: evoluir um programa que operasse em uma seqüência de símbolos já observados, de forma a produzir um símbolo de saída que maximizasse o desempenho do programa. Este desempenho era avaliado comparando o símbolo de saída com o próximo de entrada da máquina de estados, além do uso de uma função de aptidão.

Em sua proposta, uma população de máquinas de estado finitas eram expostas a um ambiente, neste caso, uma seqüência de símbolos (ou entradas para a máquina) conhecidos até o momento. Para cada máquina, à medida em que um símbolo de entrada era oferecido à máquina, cada saída era comparada com a próxima entrada. No fim deste processo, cada símbolo teria um valor associado, e o valor médio por símbolo indicaria a aptidão da máquina. As novas máquinas eram produzidas através de mutação aleatória de cada máquina. Esta mutação podia ser feita de várias formas (por exemplo, alterar uma saída, adicionar um estado, entre outros) e, normalmente, cada máquina produzia somente uma descendente [Fog90].

As máquinas com os melhores valores de aptidão tornavam-se pais na próxima geração, e este processo era repetido até que fosse solicitada uma transição com um símbolo pertencente ao ambiente, mas que ainda não houvesse sido utilizado (isto pode ser visto como a definição de um objetivo). A melhor máquina de estados seria, portanto, a máquina que fizesse o uso deste símbolo; o novo símbolo é adicionado à lista de símbolos utilizados e o processo evolutivo recomeça.

Os resultados obtidos por Fogel utilizando este procedimento em problemas de previsão, identificação e controle automático, mostraram a viabilidade de sua proposta. Mesmo assim, alguns trabalhos a criticaram, dizendo que ela só funcionava para problemas simples [Fog90]. Além disso, entre 1976 e 1985 pouca pesquisa foi feita envolvendo a PE. Isto explica o fato da PE ter ficado "esquecida"até o início da década de 90, quando foi reintroduzida e refinada por David B. Fogel [FA90]. O refinamento tornou-a praticamente idêntica a uma EE. O processo de seleção utilizava uma forma de seleção por torneio. Na seleção por torneio, um grupo de indivíduos é escolhido aleatoriamente na população, e o

melhor indivíduo deste grupo é selecionado. Este método é somente um pouco diferente da utilizada pelas EEs, e a mutação também é semelhante. A representação utilizada normalmente está atrelada ao domínio do problema, mas a representação geralmente utilizada é um vetor de tamanho fixo com valores reais. É importante citar que a PE não utiliza nenhum mecanismo de recombinação, o que implica na utilização somente de operadores de mutação.

De fato, ao compararmos a PE com as EEs, nota-se que a PE é muito parecida com as EEs sem a etapa de recombinação. Porém, é importante deixar claro que um mecanismo de seleção proporcional à adaptabilidade difere do mecanismo determinístico utilizado pelas EEs [Coe03]. Tal como as EEs, a PE simula a evolução enfatizando mais a ligação fenotípica entre os indivíduos (pais e filhos) do que a ligação genética, enfatizada pelos AGs. Ou seja, nos AGs as mudanças que são realizadas em alguma codificação do problema devem ser decodificadas e operacionalizadas de forma que os comportamentos sejam observados e avaliados. Já na PE (e EEs), os operadores atuam o mais diretamente possível no fenótipo do indivíduo de forma a alterar o comportamento do sistema.

3.3.3 A Programação Genética

A PG é um método sistemático para gerar programas a partir de uma instrução, em alto nível, que diga o que o programa deve fazer [KP03]. Esta abordagem surgiu na década de 90 como resultado dos trabalhos de John R. Koza [Koz92, Koz90, Koz89], e pode ser entendida como uma extensão dos AGs. Isto porque os indivíduos da população, candidatos a solução, não são representados como conjuntos de tamanho fixo utilizando caracteres, e.g. bits, mas sim como programas. Em suma, a PG transforma iterativamente uma população de programas de computador em uma nova geração de programas através de operadores genéticos análogos aos de outros AEs.

Os programas são geralmente representados na forma de árvores de sintaxe ao invés de código-fonte, e normalmente são implementados utilizando a linguagem de programação LISP. Esta árvore é composta de nós e arestas, sendo que os nós indicam as instruções a serem executadas e as arestas os argumentos para cada instrução. Os nós internos são normalmente chamados de funções, e as folhas da árvore de terminais. A recombinação é feita através da troca direta de sub-árvores de expressões diferentes, e a mutação é aplicada na estrutura no intuito de alterar tanto os terminais quanto as funções [KP03].

Normalmente a árvore é avaliada partindo da esquerda para a direita, utilizando busca em profundidade. Um terminal é avaliado utilizando seu valor correspondente, e a função utiliza como argumento o resultado da avaliação de seus filhos. É importante ressaltar que quanto maior forem as árvores, maior o espaço de busca. Árvores grandes limitam a efetividade da busca mas, por outro lado, árvores muito pequenas podem limitar a

capacidade da PG em encontrar soluções.

Há pouca teoria que explica a PG, mas ela é aplicada em diversas áreas do conhecimento [Coe03]. Dentre os exemplos de experimentos que atestam o seu sucesso, podemos citar o trabalho de Koza et al. [KBAK99], onde eles geraram programas que descrevem circuitos eletrônicos, utilizando a PG. Uma certa quantidade de circuitos patenteados foram redescobertos, além de terem encontrado circuitos para realizar uma tarefa bem específica, o que muitos engenheiros elétricos consideravam praticamente impossível de ser feito [Whi01].

3.3.4 Os Algoritmos Genéticos

O AG foi proposto por John Holland em 1975 [Hol75] na Universidade de Michigan, EUA, em Ann Harbor. Porém, desde a década de 60 Holland trabalhava com o objetivo de utilizar modelos evolutivos para a resolução de problemas complexos na área de computação [Hol62]. Sua motivação, em particular, era a de projetar e implementar sistemas adaptáveis e robustos, capazes de lidar com incertezas e mudanças ambientais, adaptando-se em função dos resultados obtidos a partir de sua interação com o ambiente.

A escolha do nome AG deveu-se à ênfase que foi dada inicialmente à representação e manipulação dos indivíduos da população baseando-se em seu genótipo ao invés do fenótipo, tal como nas EEs e PE. No meio dos anos 80, os AGs atingiram outras comunidades de pesquisa, como a de aprendizado de máquina e pesquisa operacional, o que contribuiu para difundir o seu uso.

A primeira proposta de Holland foi a do Algoritmo Genético Simples (em inglês, *Simple Genetic Algorithm*, SGA), que serviu de base para os AGs subseqüentes. O algoritmo inicia a partir de uma população P, gerada aleatoriamente, com *popsize* indivíduos, como mostra a Figura 3.2. Após isto, cada indivíduo é avaliado e alguns deles (ou todos) serão selecionados para serem pais no processo de *crossover*. Estes pais selecionados serão recombinados em $\frac{popsize}{2}$ "casais", gerando *popsize* indivíduos novos (somente se a recombinação acontecer para todos os casais), chamados de filhos, que possuem uma probabilidade (normalmente muito baixa) de sofrer mutação no próximo passo do algoritmo. Após o passo de mutação, cada indivíduo novo é avaliado e o algoritmo recomeça até que um critério de parada seja satisfeito. A solução será sempre o melhor indivíduo encontrado durante o processo de evolução. É relevante comentar que o processo de *crossover* tenta criar indivíduos melhores a partir dos indivíduos mais aptos da população, enquanto a mutação garante diversidade na população, evitando que a busca concentre-se em soluções ótimas, porém locais.

No SGA, as variáveis de projeto são codificadas em um conjunto de bits, que por sua vez representa um indivíduo da população, também chamado de cromossomo. Cada

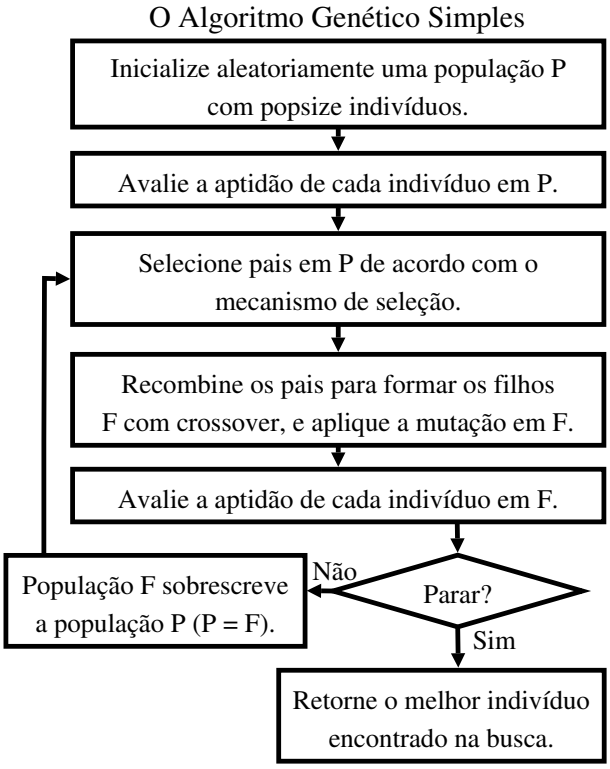

Figura 3.2: Descrição alto-nível do SGA.

indivíduo é inicializado aleatoriamente utilizando um processo que atribui para cada bit do indivíduo o valor de '0' ou '1', com probabilidade uniforme. Os componentes principais do SGA são os processos internos de seleção, *crossover* e mutação. A seleção é proporcional à aptidão, o *crossover* é de um ponto (ou simples) e a mutação bit-a-bit; estes componentes são descritos a seguir:

- *Seleção*: A seleção de pais para a recombinação é feita de acordo com a distribuição de probabilidade baseada nos valores de aptidão dos indivíduos. Este método também é conhecido como "roleta russa", uma vez que nele existem *popsize* (tamanho da população) buracos na roleta (um para cada indivíduo), cada qual com um tamanho proporcional à aptidão; os buracos maiores, que representam os indivíduos mais aptos, têm chance maior de serem selecionados. Primeiramente, calcule o valor da aptidão fit_i para cada indivíduo i da população, onde $1 \leq i \leq popsize$. Em seguida, encontre a aptidão total da população *TFit*, utilizando a equação 3.1 a seguir:

$$TFit = \sum_{i=1}^{popsize} fit_i \tag{3.1}$$

Calcule a probabilidade de seleção p_i para cada indivíduo i, tal que $p_i = \frac{fit_i}{TFit}$. O próximo passo calcula a probabilidade acumulada k_i para cada indivíduo i, utilizando a equação 3.2.

$$k_i = \sum_{j=1}^{i} p_i \tag{3.2}$$

O processo de seleção baseia-se em girar a roleta russa *popsize* vezes, e a cada vez um pai é selecionado da seguinte forma: gere aleatoriamente um número real r pertencente ao domínio [0,1]; após isto, selecione o indivíduo i tal que

$$i = \begin{cases} \text{primeiro indivíduo} & \text{se } r < k_1, \\ \textit{i-ésimo} \text{ indivíduo} & \text{onde } 2 \leq i \leq \textit{popsize} \text{ tal que } k_{i-1} < r \leq k_i. \end{cases}$$

Note que os indivíduos podem ser selecionados mais de uma vez.

- *Recombinação (crossover)*: No *crossover* de um ponto, também chamado de *crossover* simples, dois indivíduos selecionados como pais em potencial − pelo processo de seleção − trocam informações de seus subconjuntos de bits, em uma posição aleatória no indivíduo, para produzir dois novos indivíduos como mostra a Figura 3.3a.

  O *crossover* ocorre de acordo com uma probabilidade p_c, que é um parâmetro ajustável do SGA. Os indivíduos selecionados são agrupados em pares (ou "casais"), de maneira aleatória. Para cada casal, um número real r pertencente ao domínio [0,1] é gerado aleatoriamente; se $r < p_c$, então ocorrerá o *crossover*. Neste caso, gere um número inteiro *pos* aleatoriamente, pertencente ao domínio [1,m-1], onde m é o número de bits representando um indivíduo. Este número *pos* indica o ponto onde será feito o corte no conjunto de bits dos dois indivíduos, também chamado de ponto de corte, como mostra a Figura 3.3a. A partir deste momento, os indivíduos são recombinados, sendo que cada casal de pais produzirá dois indivíduos novos, chamados de filhos. Caso o *crossover* não ocorra, os filhos serão cópias idênticas dos pais.

- *Mutação*: A mutação é feita na base de bit por bit. Todos os indivíduos da população são representados utilizando bits, e todos os filhos têm a mesma chance de sofrerem mutação (mudar de '0' para '1' ou de '1' para '0'). A mutação ocorre de acordo com uma probabilidade de mutação p_m, que é também um parâmetro ajustável do SGA. Para realizar a operação, para cada indivíduo filho e para cada bit do filho, gere aleatoriamente um número real r pertencente ao domínio [0,1]; se $r < p_m$, então aplique a mutação ao bit. O resultado deste processo é um novo indivíduo, como

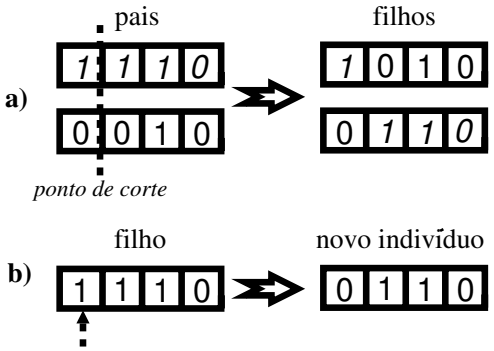

Figura 3.3: Exemplos de *crossover* (a) e mutação (b) no SGA.

mostra a Figura 3.3b.

O SGA foi escolhido para ser utilizado neste trabalho de pesquisa pelo fato dele ser a base para os outros AGs, apesar de existirem muitas outras implementações com estratégias diferentes [Cha00]. Uma variação do SGA utilizada no estudo de caso da aplicação real foi o Algoritmo Genético com *Crossover* de Dois pontos (AG-CD).

Este AG possui os mesmos processos internos do SGA. A única diferença entre eles é o processo de *crossover*. No *crossover* de dois pontos, dois indivíduos selecionados como pais em potencial (no processo de seleção) trocam informações de seus subconjuntos de bits, em duas posições aleatórias no indivíduo, para produzir dois novos indivíduos como mostra a Figura 3.4.

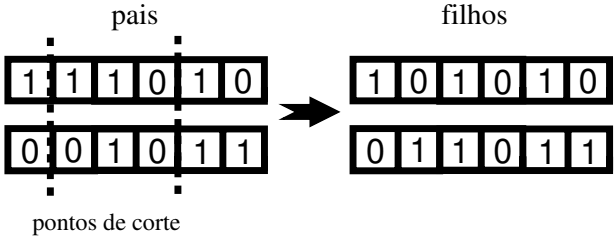

Figura 3.4: Exemplo de *crossover* de dois pontos.

Porém, no *crossover* de dois pontos selecionam-se aleatoriamente dois pontos de corte, pos_1 e pos_2, e as seções entre estes dois pontos são trocadas entre os pais. Os valores de pos_1 e pos_2 devem obrigatoriamente pertencer ao domínio [1,m-1], sendo m o número de bits representando um indivíduo.

Outras observações sobre os AGs

Há inúmeras alternativas para a implementação de um AG. Dentre estas alternativas, podemos citar a utilização da estratégia de seleção elitista, onde o melhor cromossomo de uma geração é automaticamente selecionado para fazer parte da próxima geração. Outros exemplos podem ser encontrados nos trabalhos de Coelho [Coe03] e Lacerda e Carvalho [dLdC99]. Apesar da representação dos indivíduos em um *string* de bits (codificação binária) ser a mais utilizada, há trabalhos que avaliam a utilização de outros tipos de representação [HLV98, BFM97, Mic92, Gol89]. Por exemplo, Herrera [HLV98] realiza um estudo comparativo entre AGs com codificação binária e codificação real, que por sua vez representa os indivíduos em um *string* de números reais. Ela também apresenta as inúmeras estratégias possíveis de serem utilizadas nas etapas do AG com codificação real. Em aplicações que necessitam de tratamento de valores reais, alguns pesquisadores indicam que os AGs com esta codificação têm um desempenho melhor do que os com codificação binária [Coe03]. De Jong [Jon99] afirma que os AGs não possuem um "instinto matador"do ponto de vista de otimização de funções, pelo fato deles não encontrarem uma solução muito boa tão rapidamente quanto encontram a região onde esta solução se encontra. Seu trabalho também apresenta uma perspectiva dos AGs bastante completa e interessante, incluindo alternativas para a melhoria deles para problemas de otimização.

3.4 Potencialidades e limitações

Os AEs possuem inúmeras características que os tornam uma opção interessante para a otimização de um problema. Suas potencialidades concentram-se no tratamento adequado dos sistemas sujeitos a restrições, no tratamento de problemas com sistemas complexos e espaços de busca multi-modais e/ou multi-objetivos, além do fato de não ser necessário fornecer informações relativas a derivadas (normalmente necessárias em métodos convencionais de otimização). Ele também permite que o conhecimento obtido *a priori* seja utilizado no processo, e adequam-se à implementação em paralelo e distribuídas [Coe03].

Um exemplo simples do que pode ser paralelizado é a avaliação da população do AE. Uma outra estratégia de paralelismo que é extremamente simples de se implementar e que oferece um potencial muito grande de melhorar a busca é o modelo paralelo da ilha que, basicamente, paraleliza a execução do AE particionando a população total em subpopulações (que são as ilhas). Ter subpopulações diferentes e permitir que indivíduos migrem de uma para outra age como uma forma de manutenção e exploração de diversidade na população total [Whi01].

Mas os AEs também possuem limitações. Uma delas é que seu desempenho varia de execução para execução. Sendo assim, a média de convergência sobre diversas execuções

do AE é um indicador de desempenho mais útil que uma única execução. A análise de todas as amostras do processo a cada avaliação da função objetivo é algo que deve sempre ser feito, aumentando o tempo consumido durante sua execução. Outra limitação é o esforço demandando para o ajuste dos parâmetros livres dos algoritmos.

Enfim, cabe citar o trabalho de David H. Wolpert e William G. Macready [WM97]. Eles provaram dois teoremas, chamados de *No Free Lunch*, que mostram que se um algoritmo tem um desempenho bom, em média, para uma classe de problemas, logo ele deve obrigatoriamente ter um desempenho pior, em média, no restante das classes de problemas. Sendo assim, seria insensato afirmar que um AE é superior a outro método de busca, pois o desempenho médio de qualquer par de algoritmos é idêntico quando aplicado em todos os problemas possíveis, estáticos ou dependentes de tempo. Todavia, o que se vê na prática é que os AEs são superiores a outros métodos de busca para a resolução de problemas práticos, do dia-a-dia. Estes problemas, em geral, apresentam espaços de projeto com múltiplos mínimos locais e ruídos e, neste caso, o uso dos AEs é mais apropriado [Eld98]. Isto elimina o argumento de que não é necessário desenvolver novos algoritmos, uma vez que todos os algoritmos têm o mesmo desempenho médio (afirmado pelos teoremas *No Free Lunch*). Uma das grandes motivações para o desenvolvimento de novos algoritmos é melhorar o desempenho em problemas práticos.

Capítulo 4

Algoritmos evolutivos aplicados a testes de software

No contexto de geração automática de dados utilizando a abordagem dinâmica, há uma série de trabalhos que utilizam AEs como método para gerar os dados de teste. Estes trabalhos são recentes e muitos deles utilizam o AG, sendo que a diferenças principais consistem na:

1. implementação do AG, visto que cada processo interno (seleção, *crossover* e mutação) pode ser implementado com estratégias diferentes do SGA;
2. critério de testes, que é normalmente mensurado através da análise de cobertura de uma determinada característica do programa (instruções, caminhos, decisões, etc) pelo conjunto de dados gerados;
3. e a função objetivo, que avalia a qualidade (ou aptidão) de cada candidato a solução (ou indivíduo), sendo a guia para o AE na geração dos dados de teste.

Este capítulo trará detalhes dos trabalhos que lidam especificamente com geração dinâmica de dados de teste, uma abordagem proposta em 1976 por Miller e Spooner [MS76], para o teste orientado a caminhos. Embora não sejam o foco desta pesquisa, também serão apresentados trabalhos relevantes na área de geração automática de dados de teste envolvendo outros critérios de adequabilidade como, por exemplo, teste de decisões. Vale lembrar que tanto o critério de caminhos quanto o de decisões foram descritos no Capítulo 2.

4.1 Funções-objetivo para teste orientado a caminhos

Como visto no Capítulo 2, as três abordagens principais para a geração automática de dados de testes são a aleatória, a de execução simbólica e a dinâmica. Além disso, a

geração dos dados de teste pode ser guiada utilizando basicamente dois tipos de análise, que é a análise de fluxo de dados e a de fluxo de controle.

Outro elemento importante na geração dos dados é a função objetivo. Uma função objetivo é o componente responsável pela modelagem matemática do problema que se deseja resolver em um problema de otimização. No caso do problema de teste orientado a caminhos, as principais funções-objetivo foram propostas por Lin e Yeh [LY01], Bueno e Jino [BJ02], Mansour e Salame [MS04], e Watkins e Hufnagel [WH06].

A função objetivo de Lin e Yeh, chamada de *Similarity* [LY01], recebe como entrada um caminho alvo (um caminho que deve ser coberto) e outro qualquer, e quantifica a distância entre eles. A saída é um número que mostra a similaridade entres os caminhos e que mede a qualidade do dado de teste produzido. Quanto maior a similaridade (a saída da função objetivo), melhor o dado de teste. Esta foi a função objetivo utilizada nos experimentos neste trabalho, e os detalhes de seu funcionamento encontram-se no Capítulo 6, na Seção 6.2.

A função objetivo proposta por Mansour e Salame [MS04] envolve três passos: (1) conversão de todas as condições de cada decisão do programa do formato de inequação para equação, (2) avaliação simbólica do lado esquerdo e direito de cada condição de cada decisão que pertence ao caminho que se deseja cobrir, e (3) a comparação entre os valores do lado esquerdo e direito de cada condição de cada decisão. Esta comparação foi feita utilizando uma *Hamming Distance* (HD) com peso [MS04]. A HD compara dois vetores de bits, e retorna o número de bits distintos entre os vetores, como mostra a Figura 4.1. Uma HD com peso faz o mesmo que a HD convencional, mas multiplica o resultado por um fator de peso, que pode ser calculado de diversas formas [Sth96]. Sendo assim, cada decisão terá um valor de aptidão (resultado da HD com peso), e a aptidão de um caminho será a soma dos valores de cada decisão que pertence a ele.

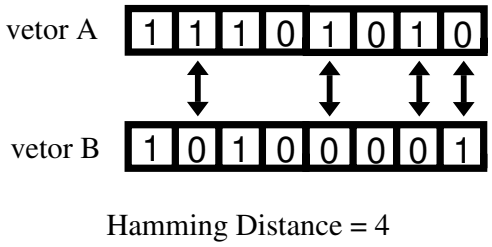

Figura 4.1: *Hamming Distance* entre dois vetores A e B.

Bueno e Jino [BJ02] propuseram uma nova função objetivo para teste orientado a caminhos, capaz de levar em consideração tanto o número de arestas em comum entre um caminho qualquer e um caminho alvo, e a decisão onde ocorreu um desvio do caminho alvo. A primeira parte da função objetivo envolve a contagem do número de arestas em comum

entre os dois caminhos, partindo da primeira aresta até a aresta onde ocorreu um desvio. Por exemplo, suponha dois caminhos C_{alvo} (caminho alvo) e $C_{coberto}$ (caminho coberto), tal que $C_{alvo} = \{a, b, c, d\}$ e $C_{coberto} = \{a, b, g, h, d\}$. Partindo da primeira aresta, estes dois caminhos possuem as arestas a e b em comum até ocorrer um desvio do caminho alvo. O valor para a primeira parte da função objetivo é o número de arestas em comum acrescido de 1. Esta soma indica o nó onde ocorreu o desvio do caminho desejado; logo, no exemplo anterior, o desvio do caminho desejado ocorreu no terceiro nó, uma vez que somente duas arestas no caminho coincidiram. A segunda parte da função objetivo analisa a decisão do nó onde ocorreu o desvio do caminho desejado, aplicando uma penalidade aos dados de teste. Quanto mais distante o dado de teste estiver de satisfazer a condição na decisão do nó, maior a penalidade. Para o cálculo das penalidades, funções de custo são associadas a todos as decisões do programa. Em suma, a função objetivo de Bueno e Jino [BJ02] visa maximizar o número de arestas em comum entre dois caminhos e minimizar a penalidade dos dados de teste que fizeram com que uma decisão não fosse satisfeita.

A probabilidade inversa de ocorrer um caminho (em inglês, *Inverse Path Probability*, IPP) [WH06] é uma versão da função objetivo proposta na década de 90 por Watkins em sua tese de doutorado. A IPP é extremamente simples, e trabalha contando quantas vezes cada caminho foi coberto até o momento. A aptidão de um indivíduo será sempre o inverso deste número. Em outras palavras, quanto mais vezes um caminho for coberto, pior será a aptidão dos indivíduos que o cobrem. Por exemplo, suponha que um caminho A tenha sido coberto por um dado de teste pela primeira vez; neste caso, a aptidão do dado de teste será $1/1 = 1$. Se o próximo dado cobrir o mesmo caminho, a aptidão dele passa a ser $1/2 = 0{,}5$, e assim sucessivamente. Sendo assim, a IPP incentiva a diversidade de caminhos, uma vez que caminhos que ocorrem menos vezes estarão mais aptos entre os indivíduos da população.

4.2 Alguns trabalhos existentes

Uma das alternativas mais utilizadas para gerar dados de teste é o AG. Outras opções incluem a Busca Tabu (em inglês, *Tabu Search*, TS) [DTB03a], o método dos gradientes [Kor90], o algoritmo de recozimento simulado (em inglês, *Simulated Annealing*, SA) [TCM98], as Estratégias Evolutivas (EEs) [AC05], a Programação Genética (PG) [EV03] e a busca aleatória, normalmente chamada de *Random-Test* (RT). Também existem muitos critérios de testes: o orientado a caminhos [WH06, AMS05, MS04, BJ02, LY01, PGGZ94, Kor90], de decisões [PHP99, Sth96], de definições e usos (*def-usos*) [FV04], de condições e decisões [AC05, MMS01] e de instruções [PHP99] são apenas alguns exemplos. A próxima seção apresenta com mais detalhes alguns trabalhos para a geração de dados de teste para a cobertura de caminhos, uma vez que este é o critério utilizado neste

trabalho de mestrado. Além disso, a Seção 4.2.2 apresenta brevemente alguns trabalhos que utilizam outros critérios.

4.2.1 Geração de dados para teste orientado a caminhos

Um exemplo clássico da abordagem dinâmica de geração de dados para teste orientado a caminhos é o trabalho de Korel [Kor90], onde ele apresentou um algoritmo determinístico para gerar dados de teste para cobrir caminhos selecionados. A abordagem dele era baseada na execução do SUT, métodos de otimização de funções e análise dinâmica de fluxo de dados. Quando alguma entrada era executada no SUT, o fluxo de execução do programa era monitorado. Caso um fluxo de execução indesejado fosse observado em uma decisão durante a execução, o método de otimização de funções era aplicado à decisão para encontrar um dado de entrada que fizesse com que o fluxo seguisse o caminho desejado. Neste contexto, a análise dinâmica de fluxo de dados era útil para indicar as variáveis mais promissoras de serem exploradas na busca pelo dado de entrada desejado. Korel utilizou a técnica de gradientes como método de otimização de funções.

A proposta deste trabalho de mestrado não faz o uso de análise dinâmica de fluxo de dados tal como na proposta de Korel [Kor90]. Ao invés disso, é feita uma análise do fluxo de controle do programa, de forma que seja possível saber quais os fluxos de execução (ou caminhos) possíveis para o programa. Sabendo os caminhos possíveis, é possível selecionar quais deles deverão ser cobertos durante a geração de dados de teste. Outra diferença é que na proposta de Korel o método de otimização atua diretamente na decisão onde ocorreu o desvio do caminho desejado, enquanto que em nossa proposta o método de otimização atua no caminho final percorrido. Korel utilizou o método dos gradientes, enquanto nossa proposta utiliza o GEO, que é um algoritmo evolutivo recente. Por outro lado, nossa proposta também utiliza a monitoração da execução utilizando como forma de saber qual caminho foi coberto por um determinado dado de teste.

Lin e Yeh [LY01] compararam o desempenho de um AG com a geração aleatória de dados de teste, utilizando como *benchmark* o programa do triângulo simplificado, que classificava um triângulo em 4 categorias: equilátero, isósceles, escaleno ou "inválido". Além disso, propuseram a *Similarity*, uma função objetivo específica para avaliar caminhos em um programa que, inclusive, foi a função selecionada para guiar o GEO e os demais algoritmos na geração de dados de teste neste trabalho de mestrado. Os autores apresentam detalhes do funcionamento da função objetivo, incluindo uma descrição passo-a-passo tanto teórica quanto prática, o que facilita e muito a compreensão do método proposto. Para a avaliação, eles compararam o AG-CD com o *Random-Test*. Os resultados mostraram a superioridade do AG-CD quando comparado ao *Random-Test*.

No entanto, o trabalho de Lin e Yeh carece de detalhes que são de suma importância

para a comparação do trabalho deles com o de outros pesquisadores: por exemplo, o domínio das variáveis de projeto não foi especificado, o que certamente faz uma grande diferença no grau de dificuldade da busca por dados de teste. A avaliação da proposta foi feita utilizando somente um programa de testes, o triângulo simplificado. Uma limitação do método é que ele não é capaz de diferenciar dados de teste que cubram um mesmo caminho. Além disso, apesar de os autores mencionarem que a função objetivo deles funcionava bem para caminhos com *loops*, nenhum programa de teste com *loops* foi avaliado, muito menos em trabalhos subseqüentes dos próprios autores ou de outros. Em suma, nossa proposta difere da destes autores devido ao AE utilizado: Lin e Yeh utilizam um AG-CD, enquanto nossa proposta utiliza o GEO, que será descrito no próximo capítulo. Além disso, nosso estudo vai muito além do que os autores propuseram pois, além de avaliar a *Similarity* para caminhos com *loops*, ele avalia um novo AE para a geração de dados de teste utilizando uma série de programas pequenos bem conhecidos, além de um com complexidade maior, muito mais linhas de código e caminhos.

Bueno e Jino [BJ02] apresentaram em 2002 uma nova função objetivo para teste orientado a caminhos. O objetivo deles era encontrar dados de teste que resolvessem restrições impostas pelas decisões ao longo de um caminho alvo. Sendo assim, um dado de teste que resolvesse uma decisão deveria continuar resolvendo todos as decisões anteriores (senão haveria um desvio do caminho alvo). Eles compararam os resultados da abordagem proposta utilizando um AG e o RT, realizando uma avaliação empírica em seis programas de teste. A abordagem proposta por eles apresentou uma série de novidades:

1. A função objetivo, que avalia a semelhança entre dois caminhos comparando suas arestas e associando funções-penalidade às decisões ao longo dos caminhos;
2. Geração de dados de teste capazes de lidar com vetores, *strings*, ponteiros, *loops* e expressões aritméticas complexas;
3. O ponto inicial da busca por dados de teste é escolhido baseado em dados de teste antigos, que chegaram mais perto de cobrir o caminho alvo. Os resultados mostraram uma redução de 53% no número de execuções do SUT;
4. Realiza um novo tratamento de construções do tipo *switch/case* (todo item indesejado é marcado com um valor relativo para incentivar a execução da decisão desejada), que antes precisavam ser convertidas para construções do tipo *if/else*.

O trabalho de Bueno e Jino [BJ02] utiliza em uma das partes de sua função objetivo uma informação semelhante à oferecida pela *Similarity*: a semelhança entre dois caminhos. Além disso, ele analisa a qualidade de cada dado de teste gerado, penalizando aqueles que se afastarem de cobrir uma decisão que faz parte do caminho. Um ponto interessante é que o trabalho descreve muito bem a função objetivo e como ela funciona. Os autores fazem considerações importantes a respeito da representação de tipos complexos de dados, bem

como definem uma forma para tratar do problema de *loops* infinitos. A função objetivo proposta por eles, apesar de apresentar resultados muito bons, não foi utilizada em nossa proposta visto que o objetivo deste trabalho é avaliar o GEO para a geração automática de dados de teste e compará-lo com outros algoritmos para a geração de dados, o que não torna a escolha da função objetivo uma prioridade. Tanto a proposta deles quanto a nossa utiliza o CFG como forma de representação e extração dos caminhos do programa. Uma diferença do trabalho deles para o nosso é que nossa proposta será avaliada não somente com programas de teste conhecidos, mas também com um programa real, muito mais complexo e com muito mais linhas de código.

Mansour e Salame [MS04] compararam um AG, o algoritmo de recozimento simulado [KJV83] e a abordagem proposta por Korel [Kor90] entre si. Os autores propuseram uma função objetivo para o teste orientado a caminhos, e avaliaram três algoritmos para a geração de dados para este critério utilizando oito programas com complexidades diferentes. Os resultados mostraram que o SA e o AG cobriram um número maior de caminhos que a abordagem de Korel, sendo que o SA foi melhor que o AG. Por outro lado, a abordagem de Korel foi a mais rápida quando os três algoritmos encontravam dados para cobrir o caminho desejado. Os autores apresentaram o tempo médio para gerar dados para um dos programas avaliados, e neste caso o KA foi 531 vezes mais rápido que o SA e 22 vezes mais rápido que o GA.

Em relação ao trabalho de Mansour e Salame [MS04], os resultados deles são apresentados individualmente para cada caminho, ao invés de uma análise focada na cobertura total que cada algoritmo alcançou. O formato de apresentação é um tanto quanto confuso e o trabalho carece de uma análise mais detalhada dos resultados obtidos. Embora a abordagem deles tenha a habilidade de gerar dados de teste que estão próximos aos limites (superior ou inferior) de cada decisão, o que aumenta as chances de revelar falhas [Cla76], o trabalho deles não apresenta detalhes dos passos da função objetivo. Além disso, a avaliação simbólica de um caminho complexo seria uma tarefa no mínimo trabalhosa, e.g. um caminho com decisões envolvendo elementos de um vetor ou comparações entre *strings*. Uma limitação que os próprios autores apresentam é que a abordagem deles só funciona para caminhos com decisões que trabalham com valores numéricos. A HD com peso que eles utilizaram foi avaliada por Sthamer [Sth96] em sua tese de doutorado, e os resultados indicaram que ela não era a melhor estratégia de pesos. Por fim, o trabalho deles não trata o problema dos caminhos que possuem *loops*. Nossa proposta difere da deles por quatro motivos: (1) nossa proposta utiliza um novo AE no processo de geração de dados de teste, o GEO; (2) a função objetivo utilizada em nossa proposta não avalia se um dado está próximo ou não dos limites de cada decisão, mas se um dado de teste cobre um caminho que se assemelha cada vez mais com um determinado caminho alvo; (3) a avaliação simbólica não é utilizada em nosso trabalho, visto que esta apresenta inúmeras

desvantagens quando aplicada a entradas complexas, como já foi visto no Capítulo 2; (4) a complexidade das decisões não interfere em nossa proposta, particularmente na função objetivo *Similarity*.

Um trabalho bem recente e muito interessante é o de Watkins e Hufnagel [WH06], pois nele é feita uma comparação entre várias funções-objetivo projetadas especificamente para teste orientado a caminhos. Eles descrevem brevemente cada uma das quatro principais funções-objetivo propostas por outros autores, apresentando suas vantagens e desvantagens além de um exemplo simples de seu funcionamento. Estas quatro funções foram descritas na Seção 4.1. Além destas, os autores propõem uma função objetivo que combina alguns artifícios da função de Lin e Yeh com a de Bueno e Jino, além de uma que utiliza somente a função penalidade de Bueno e Jino. Uma última função objetivo proposta no mesmo trabalho é chamada de *estática*, visto que todos os indíviduos terão a mesma aptidão; o objetivo disto era mostrar o impacto de uma boa função objetivo na busca por dados de teste. Para avaliar as funções-objetivo, os autores utilizaram um AG e a geração aleatória de testes em dois programas, um simples e outro complexo com muitas variáveis e caminhos. Os critérios de avaliação foram a confiabilidade (quantas vezes todos os caminhos desejados eram cobertos) e eficiência (quantas vezes o SUT foi executado, o que é equivalente ao número de dados de teste gerados). Os resultados mostraram que a função objetivo IPP e a de Bueno e Jino obtiveram o melhor desempenho no conjunto de experimentos realizados. Por outro lado, os próprios autores apresentam uma série de limitações a respeito de seu trabalho, incluindo o próprio procedimento estatístico utilizado para comparar os resultados de cada função objetivo.

Por fim, o trabalho de Watkins e Hufnagel [WH06] faz uma pequena revisão das funções-objetivo para teste orientado a caminhos e apresentou detalhes importantes a respeito dos experimentos realizados, como por exemplo, o domínio das variáveis de projeto. Além disso, as funções-objetivo foram devidamente apresentadas, inclusive com exemplos. A avaliação das abordagens em um programa com complexidade bem maior se comparada aos programas de teste comumente utilizados na literatura é um diferencial do trabalho, assim como a aplicação de um procedimento estatístico para comparar os resultados, mesmo sabendo das limitações dele. Embora Watkins e Hufnagel citaram a abordagem de Michael et al [MMS01] como uma abordagem para teste orientado a caminhos, nosso trabalho trata-o como uma abordagem voltada para a cobertura de decisões. O grande diferencial de nosso trabalho é a avaliação de um algoritmo evolutivo recente para a geração de dados de teste.

4.2.2 Geração de dados para outros critérios

Sthamer [Sth96] fez um estudo abrangente em sua tese de doutorado a respeito do AG como uma ferramenta para a geração de dados de teste para atingir cobertura total de decisões, além das inúmeras variações de seus processos internos. Este mesmo autor propôs uma abordagem interessante para o teste de *loops*. A condição utilizada por ele foi a de que um *loop* deve ser exercitado zero, uma, duas e mais de duas vezes, se possível, antes dele ser considerado "testado". Uma semelhança do trabalho dele para nossa proposta é a estratégia para o tratamento de *loops*. Esta estratégia mostrou-se simples e, ainda mais, é automaticamente incorporada pela *Similarity*, uma vez que um caminho com *loops* irá, na prática, simplesmente repetir determinadas arestas e nada mais do que isto.

Tracey et al [TCM98] geraram dados para teste caixa-preta utilizando o SA, tendo sido o primeiro trabalho a utilizar esta metaheurística para esta finalidade. A técnica proposta pelos autores visa o teste de especificação, sendo uma abordagem bem diferente da grande maioria até então publicada, cujo foco é testes caixa-branca. O objetivo dos autores é encontrar um dado de teste que viole a especificação pois, dessa forma, uma falha será revelada. Para isso, a pós-condição da especificação formal do programa que está sendo testado é negada e, caso um dado de teste satisfaça a pré-condição e a pós-condição negada, logo a especificação possui uma falha. A técnica é bem descrita e todos os experimentos e suas condições estão bem especificadas. Ela foi avaliada em dois estudos de caso, e os resultados foram satisfatórios. Por outro lado, a técnica não foi aplicada em programas mais complexos, o que pode esconder limitações da mesma.

Outros autores, Pargas et al [PHP99], utilizaram um AG para gerar dados de teste com o objetivo de atingir a cobertura de instruções e de decisões. Eles trabalharam com um grafo de dependência de controle (em inglês, *Control Dependence Graph*, CDG) para representar o programa, e geraram valores de entrada do tipo inteiro. Este grafo representa as dependências de controle em um programa, sendo que os nós representam instruções e as arestas representam dependências de controle entre instruções. A abordagem pode ser utilizada para testar programas com chamadas a procedimentos aninhados, bem como é extensível para a análise de cobertura de caminhos e definições e usos de variáveis. Baseado na abordagem, a ferramenta TGEN foi implementada e os resultados mostraram a superioridade da abordagem proposta utilizando o AG frente ao *Random-Test*.

Diferentemente da abordagem deles, nossa proposta utiliza o CFG do programa como base para extração dos caminhos possíveis do programa. De fato, bem como os autores afirmam, o CDG fornece uma informação mais precisa a respeito do programa; por outro lado, CDGs são grafos acíclicos, o que os impede de serem utilizados na avaliação de caminhos com *loops*. Uma semelhança é que a função objetivo utilizada por eles não fornece nenhuma informação que permita classificar soluções diferentes que resultem na cobertura de uma mesma instrução, decisão ou caminho. Esta é uma limitação de nossa

proposta, em virtude da utilização da *Similarity* como função objetivo.

Outro trabalho relevante é o de Michael et al [MMS01], que descreve a implementação de uma ferramenta para a geração automática de dados de teste para programas em C/C++ baseada em um AG, chamada GADGET. Eles avaliaram a eficiência de sua abordagem em um certo número de programas (nove programas de teste e programas sintéticos, cujo comportamento foi definido pelos autores), incluindo um programa cujo tamanho é significativo (em termos de número de linhas de código e complexidade) se comparado aos demais. A abordagem proposta por eles era uma implementação da abordagem de Korel [Kor90] utilizando um AG ao invés da técnica de gradientes. O foco foi a cobertura de condições e decisões (em inglês, *condition-decision coverage*), que requer que cada decisão do código seja executada **e** que cada condição pertencente à decisão seja avaliada como verdadeiro e falso, pelo menos uma vez. A avaliação da abordagem em um programa de proporções e complexidade reais agregou muito valor à comparação feita no trabalho. Esta é, inclusive, uma das motivações para a definição de um estudo de caso com uma aplicação real para a avaliação de nossa proposta. A abordagem de Michael et al apresenta a mesma limitação da abordagem de Pargas et al [PHP99] e de nossa proposta, uma vez que as funções-objetivo utilizadas não diferenciam dados de teste que cubram um mesmo caminho ou decisão.

Díaz et al [DTB03a] apresentam uma técnica de geração de dados de teste para cobertura de decisões que combina a abordagem de Korel [Kor90] com uma metaheurística chamada de Busca Tabu (em inglês, *Tabu Search*, TS) [Tra97]. Esta metaheurística é explicada sucintamente pelos autores, mas não a impede de ser entendida. A novidade neste trabalho é a utilização do TS em testes de software, uma vez que esta já é utilizada e funciona em outros problemas de otimização. Porém este trabalho não apresenta nenhum tipo de resultado que avalia a técnica proposta, o que é um ponto extremamente negativo.

Todavia, os mesmos autores apresentam em outro trabalho [DTB03b] uma ferramenta que implementa a técnica proposta, para a instrumentação e geração automática de dados de teste para a cobertura de decisões de um programa escrito em C/C++. A representação do programa é feita através de um CFG, tal como em nossa proposta. No entanto, a técnica não foi avaliada em detalhes. Aparentemente o foco do trabalho foi avaliar a ferramenta e sua vantagem com relação à economia de tempo para a instrumentação do código fonte do programa. O programa utilizado na avaliação foi o triângulo simplificado, e as informações coletadas foram o tempo consumido para a instrumentação manual e automática, bem como o tempo da execução (utilizando o TS) e geração aleatória de testes). Um ponto negativo no trabalho dos autores é que a abordagem proposta não foi comparada com outras já existentes como, por exemplo, um AG. O domínio das variáveis de projeto também não foi definido, bem como os parâmetros ajustáveis do TS. Enfim, a abordagem proposta por Díaz et al foi avaliada superficialmente, apesar de que

a utilização da metaheurística é, de fato, uma novidade na área.

A Tabela 4.1 resume os trabalhos que foram avaliados ao longo deste capítulo. Note que a novidade de nossa proposta é o uso do GEO para a geração automática de dados de teste.

Tabela 4.1: Comparação entre trabalhos de geração automática de dados de teste.

Autor(es)	Método de otimização principal	Função objetivo	Critério de testes
Korel [Kor90]	GD	Avaliação de funções associadas às decisões ao longo de um caminho, uma a uma.	Caminhos
Sthamer [Sth96]	AG	Avaliação de funções associadas às decisões.	Predicados
Pargas et al [PHP99]	AG	Avaliação de semelhança entre decisões cobertas e o CDG.	Instruções e Predicados
Lin e Yeh [LY01]	AG-CD	*Similarity*	Caminhos
Michael et al [MMS01]	AG	Avaliação de funções associadas às decisões.	Condições e decisões
Bueno e Jino [BJ02]	AG	Avaliação de semelhança dos caminhos e função penalidade associada às decisões.	Caminhos
Díaz et al [DTB03a]	TS	Avaliação de funções associadas às decisões.	Predicados
Mansour e Salame [MS04]	AG	Avaliação de funções associadas às decisões ao longo de um caminho, utilizando a HD com peso.	Caminhos
Watkins e Hufnagel [WH06]	AG	Várias, porém a novidade é a IPP.	Caminhos
Nossa proposta	**GEO**	***Similarity***	**Caminhos**

Capítulo 5

O algoritmo da Otimização Extrema Generalizada

O algoritmo da Otimização Extrema Generalizada (em inglês, *Generalized Extremal Optimization*, GEO) é uma metaheurística proposta recentemente para lidar com problemas complexos em otimização. Ele foi desenvolvido como uma generalização do método da Otimização Extrema (em inglês, *Extremal Optimization*, EO) [BP01b], de forma que pudesse ser aplicado diretamente a uma grande classe de problemas. Tanto o EO quanto o GEO são algoritmos evolutivos, e seus processos internos foram inspirados no modelo simplificado de evolução de Bak-Sneppen [BS93], proposto para mostrar a presença da Criticalidade Auto-Organizada (em inglês, *Self-Organized Criticality*, SOC) em ecossistemas.

Este capítulo apresenta primeiramente os fundamentos teóricos que deram origem ao GEO. Em seguida, seus passos e funcionamento são detalhados e explicitados através de um exemplo prático, enquanto que o final do capítulo faz considerações quanto ao tratamento de restrições pelo GEO e apresenta algumas de suas aplicações em problemas reais.

5.1 A teoria da Criticalidade Auto-Organizada

A teoria da SOC foi proposta por Bak, Tang e Wiesenfeld [BTW87], e tem sido muito utilizada para explicar o comportamento observado em uma série de sistemas complexos em vários campos da ciência [Bak96].

Esta teoria define que grandes sistemas evoluem naturalmente para um estado crítico onde uma simples mudança em um de seus elementos produz perturbações que podem atingir qualquer número de elementos do sistema. Este estado crítico é atingido pela dinâmica intrínseca ao sistema, e a probabilidade de ocorrer uma perturbação de tamanho

s é descrita por uma lei de potência na forma

$$P(s) \approx s^{-\tau} \tag{5.1}$$

onde τ é um parâmetro positivo. Observe que as perturbações pequenas têm uma probabilidade maior de ocorrer do que as grandes, mas perturbações tão grandes quanto o sistema inteiro podem ocorrer com uma probabilidade não desprezível.

Um modelo bem comum e intuitivo de representação de SOC é a criação de uma pilha de areia, ilustrada na Figura 5.1. A areia é colocada lentamente, grão a grão, em uma superfície (5.1a); no início os grãos ficam no local onde eles caíram, mas na medida em que mais grãos são adicionados, uns começam a ficar em cima dos outros formando uma pilha com uma inclinação pequena. A partir deste momento, a adição de mais grãos fará com que a inclinação da pilha fique cada vez maior e, periodicamente, grãos irão "descer"da pilha provocando pequenas avalanches (5.1b).

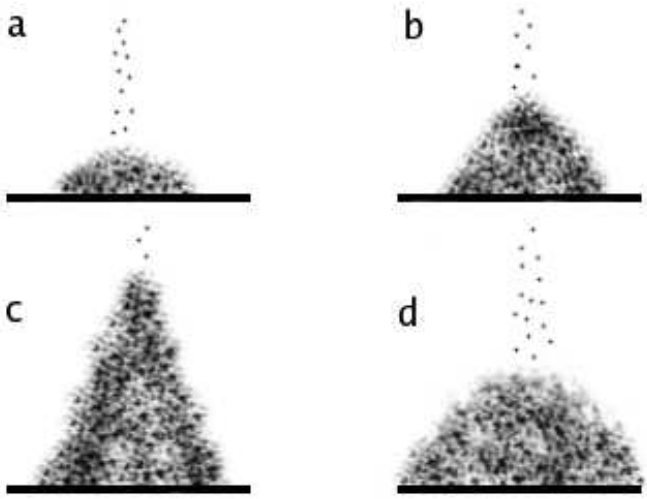

Figura 5.1: O exemplo da pilha de areia.

Com a adição de mais grãos, a inclinação da pilha pode aumentar cada vez mais e, conseqüentemente, o tamanho médio das avalanches será maior. A inclinação média da pilha pára de aumentar quando ela atinge o valor correspondente ao "ângulo crítico"(5.1c). Neste ponto a pilha está em estado crítico; sendo assim, a adição de mais um grão de areia pode provocar uma avalanche de qualquer tamanho, inclusive uma de proporções gigantescas (5.1d).

Resumidamente, a pilha de areia se auto-organizou em um estado crítico, independente das condições iniciais. No caso da pilha de areia, a freqüência de ocorrência das avalanches

e o tamanho destas foram observadas. Ao final, o comportamento do modelo da pilha de areia mostrou que a distribuição das avalanches e seus tamanhos seguiam uma lei de potência da mesma forma que a da Equação 5.1, como mostra a Figura 5.2, retirada de [BS93]. Held et al [HSK+90] fizeram vários experimentos com pilhas de areia dos mais diversos tamanhos, e observaram que elas se auto organizavam em estados críticos. No entanto, a pilha de areia com o maior tamanho dentre as analisadas apresentou oscilações nos resultados, o que sugere que a ocorrência da SOC é um efeito diretamente relacionado ao tamanho do sistema analisado.

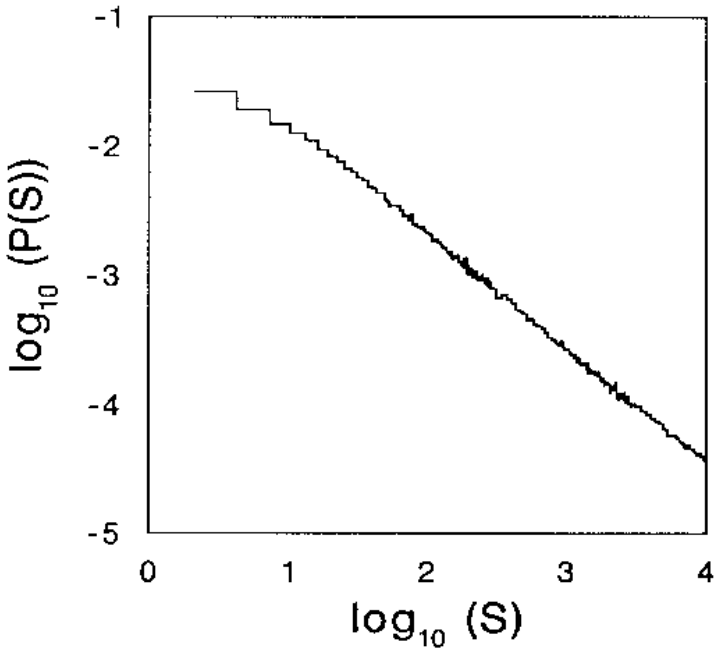

Figura 5.2: Gráfico log-log da distribuição das avalanches para o modelo da pilha de areia.

5.2 O modelo evolutivo de Bak-Sneppen

O modelo de Bak-Sneppen foi proposto para mostrar evidências de SOC na evolução das espécies. Gould e Eldredge [GE77] haviam mostrado que a evolução biológica ocorria de acordo com um grande número eventos intermitentes, seguido de longos períodos de estabilidade, e Raup [Rau86] observou que a extinção de espécies era um episódio que poderia ocorrer em todas as escalas [BS93]. Este comportamento recebeu o nome de equilíbrio pontuado [GE93], e define que a maioria das espécies que se reproduzem sexuadamente mostrarão pouca ou quase nenhuma mudança, em termos de evolução, ao longo de sua história. Quando a evolução ocorre, ela é esporádica e relativamente rápida

quando comparada com a duração total da espécie na terra.

Os experimentos de Bak-Sneppen com seu modelo mostraram a presença de equilíbrio pontuado. Nele, as espécies são representadas lado a lado e, para cada uma delas, é associado um valor de aptidão no domínio [0,1]. A evolução delas é simulada forçando a espécie menos adaptada — aquela com o pior valor de aptidão — e seus vizinhos a mudar. A Figura 5.3 mostra n espécies e define a relação de vizinhança: e_n é vizinho de e_{n-1} e e_1; e_1 é vizinho de e_n e e_2, e assim sucessivamente. A mudança é feita atribuindo-se aleatoriamente novos valores de aptidão às espécies envolvidas, e seu resultado pode ser a evolução ou extinção de qualquer uma delas, sendo que as novas espécies não terão necessariamente valores de aptidão melhores que os das espécies antigas.

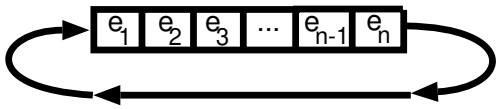

Figura 5.3: Representação das espécies no modelo de Bak-Sneppen.

Este ecossistema simulado é iniciado com os valores de aptidão das espécies distribuídos uniformemente no domínio [0,1]. Como a espécie menos adaptada é constantemente forçada a mudar, a aptidão média do ecossistema aumenta e, eventualmente algum tempo depois do início do processo, todas as espécies possuirão uma aptidão acima de um "nível crítico". Apesar disso, como até mesmo as espécies boas são forçadas a mudar (caso elas sejam vizinhas da espécie menos adaptada), um certo número de espécies pode ficar abaixo do nível crítico de tempos em tempos. Ou seja, o equilíbrio de uma ou mais espécies (estando acima do nível crítico em um estado que pode ser alterado a qualquer momento) é pontuado por avalanches, cuja ocorrência é descrita por uma lei de potência [BS93].

Uma heurística de otimização baseada no modelo de Bak-Sneppen poderia evoluir soluções rapidamente, mudando sistematicamente os piores indivíduos e preservando ao mesmo tempo, ao longo do processo de busca, a possibilidade de investigar regiões diferentes do espaço de projeto através das avalanches.

Baseado no modelo de Bak-Sneppen, Boettcher e Percus desenvolveram o algoritmo EO para abordar problemas difíceis em otimização combinatória. Este algoritmo apresentou um desempenho bom em problemas como o do biparticionamento de grafos [BP01a]. Neste problema, temos um grafo com n vértices (sendo n um número par) e alguns pares de vértices são conectados por uma aresta. O problema consiste então em dividir os vértices em dois conjuntos de mesmo tamanho $n/2$, minimizando o número de arestas m (tamanho do corte) que conectam ambos os conjuntos. Apesar do desempenho bom neste tipo de problema, em outros problemas a busca do EO tornou-se exclusivamente determinística [BP01b]. Além disso, como os próprios autores destacaram, o EO possui

problemas que restringem sua aplicação a uma grande classe de problemas. Talvez o maior deles é que uma definição geral para associar os índices de adaptabilidade às variáveis de projeto pode se tornar uma tarefa ambígua ou até mesmo impossível [BP01b], uma vez que para cada problema esta associação pode ser feita de formas diferentes.

5.3 Entendendo melhor o GEO

O GEO foi proposto para ser facilmente aplicado a uma grande classe de problemas com qualquer tipo de variável de projeto, seja contínua, discreta ou uma combinação delas, em um espaço de projeto multimodal ou mesmo descontínuo e sujeito a qualquer tipo de restrição.

As variáveis de projeto são codificadas em um *string* binário, tal como no SGA. No entanto, o GEO associa um valor de aptidão a cada bit do *string*, também chamado de espécie, ao invés de associá-lo ao *string* binário por inteiro (como no SGA) ou a cada variável de projeto (como no EO). A Figura 5.4 mostra as variáveis x e y codificadas em um *string* representado por quatro espécies no GEO, duas espécies no EO e por um indivíduo no SGA. Cada *string* representa uma solução para o problema. No SGA há uma população de n *strings* com m bits, enquanto que no GEO há somente um único *string*, com uma população de m bits. No EO, há um *string* de valores reais, um para cada variável de projeto.

Observe que no GEO cada bit é considerado uma espécie de uma população de espécies, visto que a mutação dele (mudança de '0' para '1' ou de '1' para '0') produz com os outros uma configuração diferente, que por sua vez é uma solução diferente para o problema que está sendo avaliado. Sendo assim, um *string* binário com n bits terá n espécies; por exemplo, suponha o *string* $\{000\}$. A mutação do primeiro bit produzirá $\{\mathbf{1}00\}$, a do segundo, $\{0\mathbf{1}0\}$, e a do terceiro, $\{00\mathbf{1}\}$. Note que toda mutação de um bit produziu uma configuração diferente, que é uma solução diferente. Em uma aplicação real, o número de bits necessários para representar cada variável de projeto depende de seu domínio e precisão.

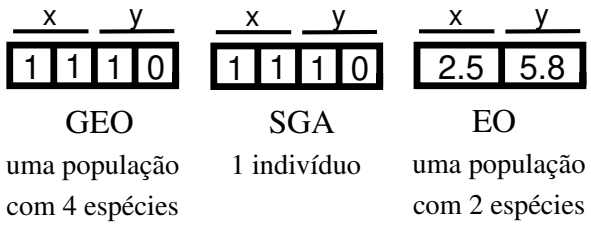

Figura 5.4: Representação das espécies e indivíduos no GEO, EO e SGA.

No primeiro passo do GEO, o *string* binário com m bits (ou espécies) é inicializado aleatoriamente. Para avaliar a qualidade de cada solução contida no *string*, o ganho ou perda que a função objetivo tem com a mutação de cada bit (chamado de índice de adaptabilidade $\Delta V_i = V_i - V_{melhor}$, $i = 1, ..., m$) é calculado forçando a mutação dos bits do *string*, um de cada vez. Após o cálculo de cada ΔV, o bit é restaurado para seu estado inicial. No próximo passo, os bits são ordenados de acordo com seus valores de ΔV, começando em 1 para o pior ΔV, até m, para o melhor. A maneira pela qual eles são ordenados depende do tipo de problema de otimização que está sendo avaliado. Em problemas de minimização, o menor valor de ΔV ocupa a primeira posição na ordenação, e vice-versa para os de maximização.

Após a ordenação dos bits, um deles é escolhido com probabilidade uniforme para ser o candidato a sofrer mutação. Observe que, sendo um candidato, ele pode sofrer mutação ou não. O bit escolhido sofrerá mutação de acordo com a seguinte regra: um número real e aleatório RAN é gerado no domínio [0,1]; a probabilidade de mutação do bit escolhido i é calculada com a equação $P(i) = k_i^{-\tau}$, onde k_i é a posição de i na ordenação, e τ um parâmetro ajustável e positivo; no próximo passo, o número RAN é comparado com $P(i)$ e, se $P(i) \geq RAN$, o bit i sofre a mutação; caso contrário, outro é escolhido para ser o candidato para sofrer mutação, e o processo é repetido até que um sofra mutação. O algoritmo repete estes passos até que um critério de parada seja satisfeito, e a melhor configuração de bits será a solução para o problema. A descrição detalhada dos passos do GEO, adaptado de [Sou02], encontra-se a seguir:

Passo 1. Inicialize aleatoriamente um *string* binário (configuração de bits) de tamanho L que codifique N variáveis com tamanho de bits l_j, onde $j = 1, N$. Para a configuração inicial de bits C, calcule o valor da função objetivo V e faça $C_{melhor} = C$ e $V_{melhor} = V$.

Passo 2. Para cada bit i da configuração de bits, em uma dada iteração,

a) mude o valor do bit (de '0' para '1' ou '1' para '0') e calcule o valor da função objetivo V_i da configuração de bits C_i,

b) atribua ao bit o índice de adaptabilidade $\Delta V_i = V_i - V_{melhor}$, que indica o ganho (ou perda) relativa que o bit tem ao ser invertido, comparado ao melhor valor da função objetivo encontrado até o momento, e

c) retorne o bit ao seu valor original.

Passo 3. Ordene os bits de acordo com seus índices de adaptabilidade, de $k = 1$ para o bit menos adaptado até $k = L$ para o mais adaptado. Em um problema de minimização, valores altos de ΔV_i ocuparão as últimas posições na ordenação, e vice-versa para problemas de maximização. Se dois ou mais bits possuirem o mesmo índice de adaptabilidade, ordene-os aleatoriamente com distribuição uniforme.

Passo 4. Escolha com probabilidade uniforme um bit candidato i para sofrer mutação. Gere um número aleatório RAN com probabilidade uniforme no domínio [0,1]. Se $P_i(k) = k^{-\tau}$ for igual ou maior que RAN, o bit é modificado. Senão, escolha um novo bit candidato, e repita o processo até que um bit seja modificado.

Passo 5. Atribua $C = C_i$ e $V = V_i$, onde i é o bit confirmado para sofrer mutação no passo 4.

Passo 6. Repita os passos de 2 a 5 até que um critério de parada seja satisfeito.

Passo 7. Retorne C_{melhor} e V_{melhor} encontrado na busca.

Em uma variação do GEO canônico descrito acima, chamada de GEO$_{var}$, os bits são ordenados separadamente para cada *substring* que codifica cada uma das N variáveis de projeto, e N bits, um para cada variável, sofrem mutação em cada iteração do algoritmo.

O fluxograma da Figura 5.5, retirado de Sousa et al. [SRPG03], mostra as características principais do GEO e sua variação, GEO$_{var}$. Neste fluxograma, $^*l_j$ é o número de bits de cada variável de projeto j, com $j = 1, N$.

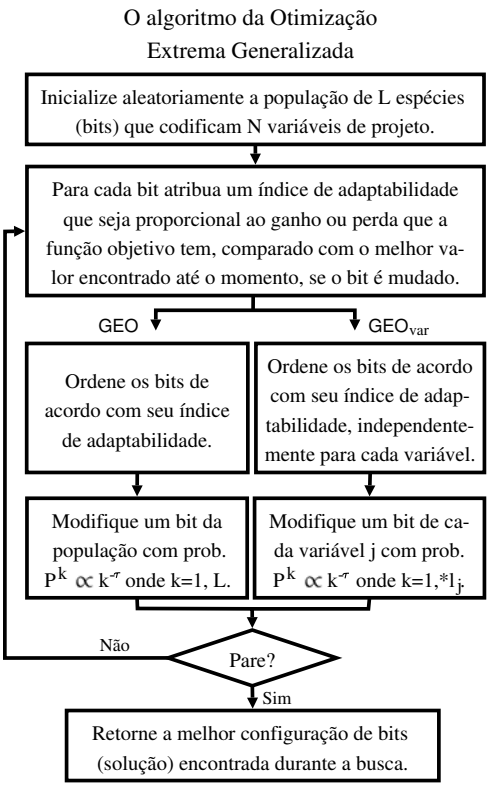

Figura 5.5: Fluxograma comparativo entre o GEO e GEO$_{var}$.

O parâmetro positivo τ é o único parâmetro ajustável do GEO. Isto dá ao GEO uma vantagem *a priori* em comparação a outros algoritmos estocásticos, uma vez que

normalmente eles têm mais parâmetros a serem ajustados, o que torna o processo de ajuste mais demorado. O valor de τ influencia na forma como é feita a busca por soluções viáveis. Se $\tau \to \infty$, somente o primeiro bit da ordenação sofrerá mutação em cada iteração do algoritmo, o que é equivalente a uma busca totalmente determinística. Por outro lado, se $\tau \to 0$, qualquer bit escolhido como candidato (estando ou não em uma boa posição na ordenação) sofrerá mutação. Analogamente, este caso é equivalente a uma busca completamente aleatória por soluções. De acordo com os trabalhos anteriores com o GEO, o valor de τ que traz os melhores resultados geralmente está no domínio [0,10] [AMS05, SVR04, SRPG03].

5.3.1 Exemplo prático de aplicação em uma função matemática

A primeira decisão a ser tomada em uma implementação prática do GEO é definir o número de bits necessários para representar cada variável de projeto. Este número depende do domínio das variáveis e sua precisão. Para variáveis contínuas, o número mínimo de bits m necessários para alcançar uma certa precisão p é dado pela Inequação a seguir

$$2^m \geq \left[\frac{\left(x_j^u - x_j^l\right)}{p} + 1 \right] \tag{5.2}$$

onde x_j^l e x_j^u são os limites inferior e superior, respectivamente, da variável j, onde $j = 1, N$, e p é a precisão desejada. Logo, dados a precisão p e o valor mínimo e máximo para uma variável de projeto x_j, é possível obter o número mínimo de bits l_j para representá-la no GEO. Observe que, caso o valor de m (obtido pela Inequação 5.2) não seja um valor inteiro, l_j passa a ser o próximo número inteiro imediatamente superior à m. O valor real de cada variável de projeto é obtido pela Equação 5.3,

$$x_j = x_j^l + \left(x_j^u - x_j^l\right) \cdot \left[\frac{I_j}{(2^{l_j} - 1)} \right] \tag{5.3}$$

onde I_j é o número inteiro obtido na transformação da variável j de sua representação binária para decimal, e l_j o número de bits necessários para representar j.

Além disso, o valor de τ deve ser definido antes da busca por soluções ser iniciada. O melhor valor para este parâmetro varia de acordo com o problema, sendo encontrado através de um processo de ajuste.

Para tornar claro o funcionamento do GEO, suponha um problema de minimização

(baseado no exemplo apresentado por Sousa [Sou02]) cuja função objetivo é

$$F(x) = \sum_{j=1}^{2} x_j^3$$

onde $-5,1 \leq x_j \leq 5,1$ e todas as variáveis têm precisão de 0,1. Através da Inequação 5.2 obtém-se $m = 6,6865$. Como o número mínimo de bits para representar uma variável deve ser um valor inteiro, $l_j = 7$.

Primeiramente, é gerada uma seqüência aleatória de bits que codifica as variáveis (passo 1 na Tabela 5.1). Esta é a melhor solução para o problema até o momento, logo $V_{melhor} = V$, onde V é o valor da função objetivo para a configuração de bits atual. No passo 2, para cada bit i da configuração, realize a mutação (alteração de '0' para '1', ou de '1' para '0'), calcule o valor da função objetivo V_i para a nova configuração de bits, e atribua ao bit i o índice de adaptabilidade ΔV_i, sendo que $\Delta V_i = V_i - V_{melhor}$. É importante ressaltar que após o cálculo de ΔV_i, o bit i retorna ao seu valor original. A Figura 5.6 mostra a função objetivo e seu valor para cada um dos 14 indivíduos da população inicial de bits. De acordo com a Tabela 5.1, o índice de adaptabilidade para o primeiro bit, $i = 1$, é $2,36$ e para o último, $i = 14$, é $64,41$. Estes números indicam que a mutação do primeiro bit da configuração representa uma redução maior no valor de V_{melhor} do que a mutação do último bit; em outras palavras, o último bit está mais bem adaptado do que o primeiro.

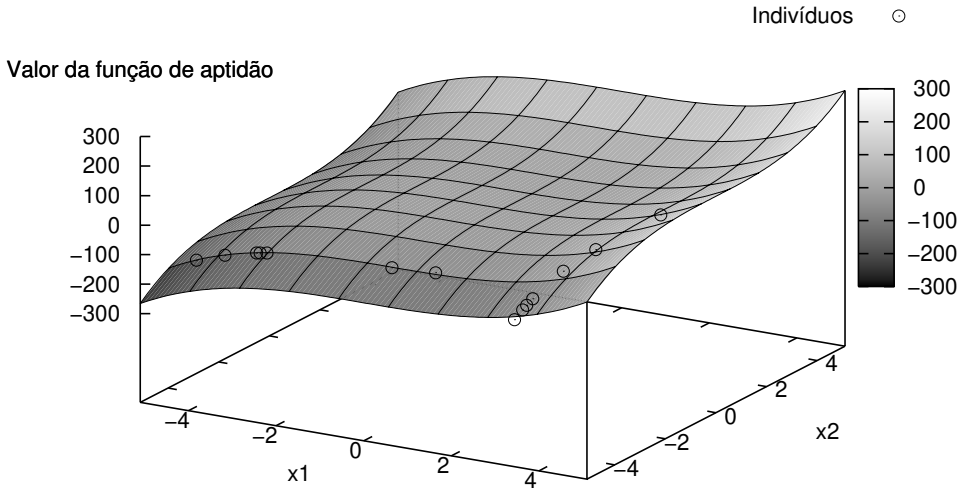

Figura 5.6: Valor da função objetivo para os indivíduos da população.

Tabela 5.1: Exemplo numérico simples dos passos do GEO.

Passo	Valor binário[a]	x_1[b]	x_2[b]	V	V_i	V_{melhor}	ΔV_i	k
1	00011000111000	$-3,17$	$-3,98$	$-94,76$	–	$-94,76$	–	–
2	***1***0011000111000	$-3,09$	$-3,98$	$-94,76$	$-92,40$	$-94,76$	$2,36$	–
	*01**1***011000111000	$-3,01$	$-3,98$	$-94,76$	$-90,16$	$-94,76$	$4,61$	–
	*001**1***11000111000	$-2,85$	$-3,98$	$-94,76$	$-86,01$	$-94,76$	$8,75$	–
	*000**0***1000111000	$-3,81$	$-3,98$	$-94,76$	$-118,36$	$-94,76$	$-23,59$	–
	*0001**0***000111000	$-4,46$	$-3,98$	$-94,76$	$-151,40$	$-94,76$	$-56,64$	–
	*00011**1***0111000	$-0,60$	$-3,98$	$-94,76$	$-63,05$	$-94,76$	$31,71$	–
	*000110**1***0111000	$1,97$	$-3,98$	$-94,76$	$-55,22$	$-94,76$	$39,54$	–
	*0001100***1**111000	$-3,17$	$-3,90$	$-94,76$	$-91,03$	$-94,76$	$3,73$	–
	*00011000***0**11000	$-3,17$	$-4,14$	$-94,76$	$-102,69$	$-94,76$	$-7,93$	–
	*000110001***0**1000	$-3,17$	$-4,30$	$-94,76$	$-111,26$	$-94,76$	$-16,50$	–
	*0001100011***0**000	$-3,17$	$-4,62$	$-94,76$	$-130,42$	$-94,76$	$-35,65$	–
	*0001100011***1**00	$-3,17$	$-2,69$	$-94,76$	$-51,41$	$-94,76$	$43,36$	–
	*0001100011***1**0	$-3,17$	$-1,41$	$-94,76$	$-34,71$	$-94,76$	$60,06$	–
	*0001100011***1**1	$-3,17$	$1,16$	$-94,76$	$-30,35$	$-94,76$	$64,41$	–
3	*0001**0***000111000	$-4,46$	$-3,98$	$-94,76$	$-151,40$	$-94,76$	$-56,64$	1
	Ordene os bits de acordo com seus índices de adaptabilidade							
	*0001100011***1**1	$-3,17$	$1,16$	$-94,76$	$-30,35$	$-94,76$	$64,41$	14
4	*000**0***1000111000	$-3,81$	$-3,98$	$-94,76$	$-118,36$	$-94,76$	$-23,59$	3
5	*000**0***1000111000	$-3,81$	$-3,98$	$-118,36$	$-118,36$	$-94,76$	$-23,59$	3
6	Repita os passos $2-5$ até que o critério de parada definido seja satisfeito							
7	Retorne a configuração de bits associada a V_{melhor} e seu valor							

[a]Os bits em itálico correspondem a x_1 e os outros a x_2. O bit menos significativo de cada variável encontra-se à esquerda do subconjunto de bits que representa cada variável. Os bits numerados em i são contados sempre da esquerda para a direita.

[b]As variáveis foram arredondadas para o número real mais próximo, dentro de sua precisão.

No passo 3, todos os bits são ordenados de acordo com seus índices de adaptabilidade definidos no passo 2. O bit i com o menor valor para ΔV_i (no caso deste exemplo, o bit $i = 5$ é o menos adaptado, portanto ocupa a primeira posição na ordenação, $k = 1$, enquanto que o bit i com o maior valor para ΔV_i, $i = 14$, é o mais adaptado e ocupa a última posição na ordenação, $k = 14$). No passo 4, um bit é escolhido como candidato a sofrer mutação. Neste exemplo o quarto bit da configuração, que ocupa a terceira posição na ordenação ($k = 3$) foi confirmado para sofrer a mutação.

Note que, como $k = 3$, este bit não é o menos adaptado da configuração, o que deixa claro a possibilidade de mutação de um bit que seja mais adaptado que outros. Para este exemplo, o valor utilizado para τ foi 1, enquanto que o número aleatório RAN gerado no passo 4 foi 0,29. Este valor de τ significa que quando o quarto bit da configuração foi escolhido aleatoriamente para sofrer mutação, este tinha uma probabilidade de $\frac{1}{3} = 0,33$

de sofrer realmente a mutação[1]. Por exemplo, se o quinto bit da configuração ($k = 1$) fosse escolhido como candidato a sofrer mutação, a mutação seria feita com uma probabilidade de 100%.

5.3.2 Tratamento de restrições

As restrições laterais no GEO, que definem o intervalo de variação (de um valor mínimo até um valor máximo) para cada variável de projeto, são incorporadas diretamente quando estas são codificadas na forma binária, visto que uma das bases para o cálculo do número de bits necessários para codificar cada variável de projeto é o domínio das variáveis, como mostrado na Inequação 5.2. As restrições de igualdade e desigualdade, também descritas no Capítulo 1, podem ser incorporadas facilmente no algoritmo. Para isto, basta atribuir um valor alto de adaptabilidade para o bit que, quando sofrer mutação, viole qualquer uma destas duas restrições e leve a uma região inviável do espaço de projeto.

Dessa forma, os bits mais adaptados dentre todos serão aqueles que produzam configurações que levem a regiões inviáveis do espaço de projeto. Logo, no passo 4 do algoritmo, a probabilidade destes bits sofrerem mutação será extremamente baixa, evitando que o algoritmo produza soluções inviáveis. Apesar disso, o algoritmo pode caminhar por regiões inviáveis do espaço de projeto, visto que nenhum bit é impedido de sofrer mutação, mesmo com probabilidade muito baixa. Isto oferece ao GEO flexibilidade, permitindo que ele seja aplicado a espaços de projeto que apresentem regiões viáveis, porém desconexas [SRPG03]. Analogamente, nada impede que o GEO inicie sua busca a partir de uma solução inviável; esta, por sua vez, será substituída pelo primeiro valor de solução viável que for encontrado na busca.

5.3.3 Aplicações do GEO

A eficácia do GEO tem sido comprovada através de vários trabalhos publicados desde a época em que ele foi proposto, em 2002. Inicialmente, ele foi comparado com outras metaheurísticas para a otimização de um conjunto de funções-teste, apresentando resultados bastante competitivos [Sou02, SR02]. O projeto de um perfil de asa para planador foi a primeira aplicação em um problema real, e os resultados foram bastante satisfatórios [Sou02, SRPG03]. Galski et al. [GSR05b] apresentam os primeiros resultados da aplicação do GEO em um problema de otimização discreto e estrutural que, neste caso, foi o problema de estrutura de 10 barras. Este problema consiste em aplicar duas cargas a uma estrutura, cada uma em um ponto específico, com o objetivo de minimizar o peso total dela. As variáveis de projeto são as áreas das 10 barras, sendo que há restrições

[1]Neste caso, o bit foi confirmado a sofrer mutação pois $P(i) = k_i^{-\tau} = 3^{-1} = \frac{1}{3} \geq 0,29 = RAN$.

quanto à tensão e compressão nas barras, bem como no deslocamento dos pontos onde as cargas são aplicadas.

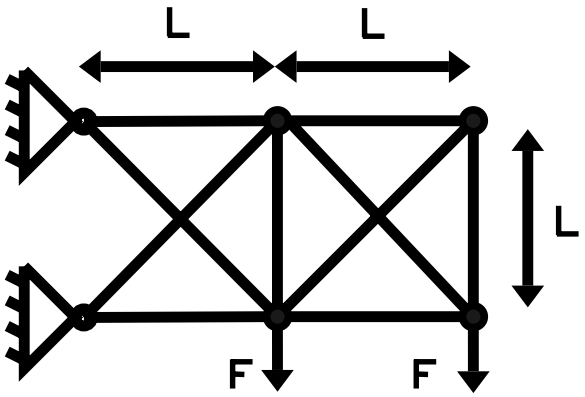

Figura 5.7: Problema estrutural das 10 barras.

Neste trabalho, o GEO é comparado com outros sete resultados — retirados da literatura — para o problema, incluindo os algoritmos genéticos (AGs) e o *Simulated Annealing* (SA). Apesar do resultado do GEO não ter sido o melhor dentre os algoritmos analisados, seu desempenho foi superior a implementações melhoradas do SGA. Além disso, seu resultado se aproximou ao de uma implementação específica do AG e também do SA que, por sua vez, foi o algoritmo que obteve o melhor resultado para este problema. Um trabalho recente aplicou o GEO na resolução do problema de estrutura de barras tridimensional (25 barras), e a solução obtida por ele foi a melhor dentre todas as encontradas na literatura por outros algoritmos [ST05]. Uma outra aplicação é detalhada por Sousa et al. [SRSS05], onde o GEO foi utilizado para a solução de um problema inverso de estimar propriedades radioativas.

Outra aplicação real do GEO é descrita por Sousa et al. [SVR04], onde ele é utilizado com sucesso no projeto de otimização do peso total de um tubo de calor para controle térmico de satélite. Um tubo de calor é um dispositivo térmico usado para transferir grandes quantidades de calor por longas distâncias com uma variação mínima de temperatura, sendo utilizado em satélites e veículos espaciais. Esta aplicação apresenta uma série de dificuldades como, por exemplo, a existência de variáveis de projeto com interações fortemente não-lineares e sujeitas a múltiplas restrições. Além desta, outra aplicação é apresentada por Galski et al. [GSRM04b], onde o objetivo é o projeto ótimo de uma configuração simplificada do subsistema de controle térmico da Plataforma Multi-Missão brasileira, em desenvolvimento no INPE. Esta, por sua vez, é uma plataforma espacial que atende a várias necessidades, podendo ser utilizada em diferentes missões como, por

exemplo, observação da Terra, além de coleta de dados científicos ou meteorológicos. No projeto térmico de uma plataforma espacial, a temperatura dos elementos que a compõe deve ser mantida dentro dos valores necessários, o que não é uma tarefa trivial se realizada manualmente.

Uma proposta recente é a versão multi-objetivo do GEO, chamada M-GEO[GSR05a]. Este foi utilizado com sucesso no projeto ótimo de uma pequena constelação de satélites *Multi-Application Purpose Synthetic Aperture Radar* (MAPSAR). O MAPSAR é um programa cooperativo de satélites envolvendo o Brasil e Alemanha cujo objetivo é avaliar, gerenciar e monitorar os recursos naturais da Terra. Considerando a importância de monitorar remotamente um desastre em uma região pouco povoada como a Amazônia, o M-GEO é utilizado para responder qual a melhor configuração de uma constelação de satélites de forma a obter o menor tempo possível de revisitação, mantendo um dos satélites na órbita original para não perder o recurso de cobertura global.

Capítulo 6

GEO aplicado a testes de software

Como foi visto no Capítulo 4, a automação de testes utilizando metaheurísticas para a geração de dados de teste é um tópico de interesse da comunidade científica. É importante avaliar outras metaheurísticas para a geração de dados, mesmo porque praticamente todos os trabalhos utilizam o algoritmo genético (AG) como metaheurística. Os inúmeros resultados apresentados em diversos trabalhos confirmaram a utilidade do AG para esta função. Há também outros trabalhos que utilizam a técnica de gradientes [Kor90], assim como o algoritmo de recozimento simulado [KJV83] e a Busca Tabu [Tra97].

Tendo em vista a importância de avaliar novas metaheurísticas na automação de testes, a proposta deste trabalho é avaliar pela primeira vez uma proposta recente, o algoritmo da Otimização Extrema Generalizada, para a geração automática de dados de teste. Além disso, o seu desempenho será comparado com o de um Algoritmo Genético Simples e a geração aleatória de dados.

A abordagem escolhida para a geração de dados é a dinâmica, que utiliza métodos de otimização de funções, monitoramento e execução do SUT. A técnica de testes utilizada ao longo deste trabalho é a técnica caixa-branca, também conhecida como estrutural. O código-fonte do SUT é necessário para que o mesmo seja instrumentado.

Em nossa proposta, o GEO será avaliado para a geração de dados para um software, com o objetivo de satisfazer o critério de caminhos, o que remete à proposta do teste de caminhos, apresentada no Capítulo 2. Este teste apresenta inúmeras dificuldades principalmente no que diz respeito aos caminhos com *loops*, e avaliar a geração de dados através do GEO para um SUT com *loops* foi um ponto importante para a escolha deste critério. Mesmo sendo um critério mais "forte"que os demais, é importante lembrar que o objetivo deste trabalho não é realizar teste de caminhos, uma vez que este tipo de teste não é possível pelas razões citadas na Seção 2.1.2 do Capítulo 2. O que será feito é a seleção aleatória de um subconjunto de caminhos do SUT que satisfaçam aos requisitos de teste desejados, e gerar dados na tentativa de cobrir este subconjunto pré-selecionado.

É importante lembrar que o mecanismo de seleção deste subconjunto não é o foco desta pesquisa, o que justifica a escolha por um método de seleção aleatório.

A eficiência do GEO será avaliada aplicando-o em um conjunto de estudos de caso relativamente simples, mas que incluem programas com e sem *loops*. Além disso, será feito um estudo de caso com uma aplicação real (implicando em um número muito maior de caminhos) com a finalidade de validar o GEO para a geração automática de dados para teste de caminhos. Em ambos os conjuntos de experimentos, o objetivo será a cobertura total de um subconjunto de caminhos do SUT. O algoritmo genético simples e a geração aleatória de dados de teste, chamada de *Random-Test*, também serão aplicadas nos conjuntos de experimentos; sendo assim, o GEO poderá ser comparado com algumas das técnicas normalmente utilizadas nos trabalhos científicos da área até o momento.

A seção a seguir descreve como estão organizados todos os recursos envolvidos na geração automática de dados de testes. Ela também descreve os passos necessários para iniciar a geração dos dados. A Seção 6.3 descreve como ocorre a geração dos dados. Os detalhes a respeito do GEO encontram-se no Capítulo 5, e a *Similarity* é descrita na Seção 6.2. Finalmente, a última seção deste capítulo descreve como ocorre a geração dos dados.

6.1 Recursos envolvidos e configuração da geração dos dados de teste

Os recursos envolvidos na geração dos dados de teste estão representados na Figura 6.1. Estes são os componentes principais para a implementação de uma ferramenta com suporte à geração automática de dados para teste de caminhos. É necessário um módulo para a instrumentação de código-fonte e compilação, mesclado à construção e instrumentação do CFG relativo ao programa que será testado[1]. Estas funções foram delegadas ao Instrumentador. Com o CFG instrumentado em mãos, é possível extrair uma lista de caminhos-alvo do programa que será testado através de um outro módulo chamado de Extrator de caminhos.

O Controlador é responsável por coordenar o início e o término do processo de geração de dados de teste. Caso as entradas requeridas (código-fonte instrumentado e compilado, e lista de caminhos-alvo) estiverem disponíveis, o Controlador notifica o módulo de monitoramento (Monitor), que por sua vez notifica o GEO, iniciando a geração dos dados de teste. O GEO interage sempre com o módulo Executor, responsável pela execução do programa instrumentado, e o módulo Similarity, que contém a implementação da função

[1] A instrumentação do CFG (particularmente, suas arestas) é necessária para que seja possível extrair caminhos a partir dele.

objetivo escolhida para avaliar os caminhos cobertos pelos dados de teste. O módulo Monitor é o responsável por coletar informações a respeito da execução do programa a partir do módulo Executor, elaborando uma lista de caminhos-alvo cobertos e os respectivos dados de teste, que será a saída ao final do processo.

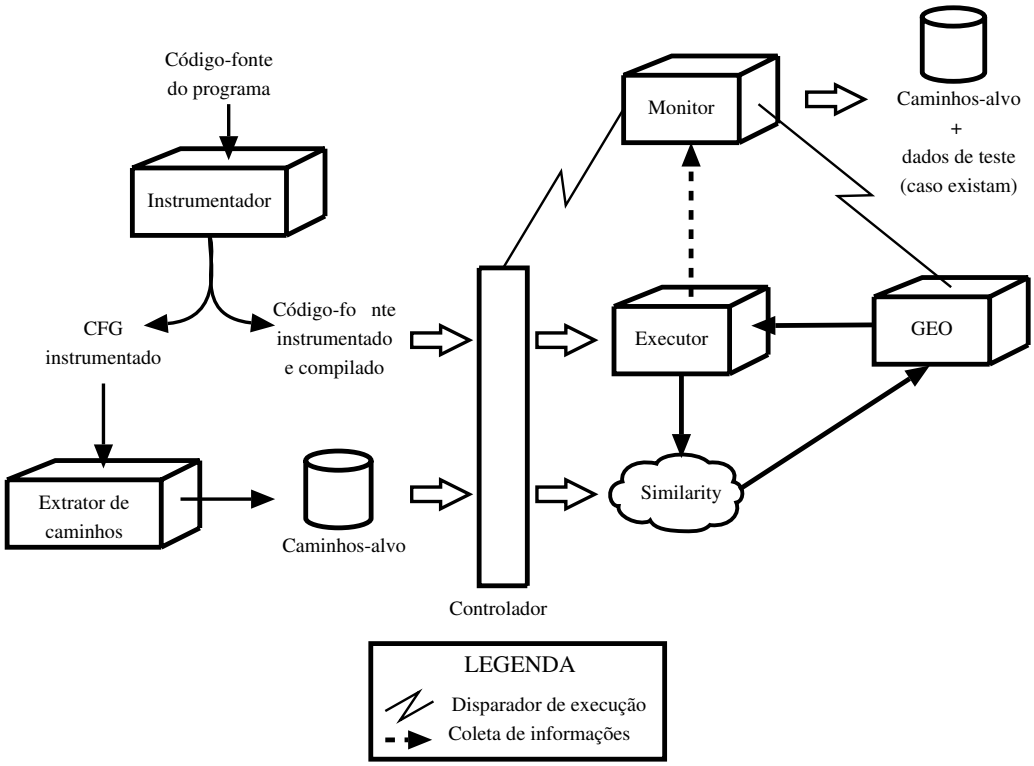

Figura 6.1: Recursos envolvidos.

Antes de iniciar o processo para gerar dados de teste para cobrir os caminhos de um SUT, o usuário deve executar as seguintes tarefas:

1. Construir o CFG a partir do SUT, instrumentar suas arestas, e instrumentar o código-fonte do SUT de acordo;
2. Definir um subconjunto de caminhos do SUT a serem cobertos (caminhos-alvo). A seleção de caminhos foi feita de modo aleatório, visto que este não é o objeto de estudo deste trabalho;
3. Definir o tipo das variáveis de projeto (ou variáveis de entrada), podendo ser **int**, **long**, **float**, **double**, **char**, **string** ou **boolean**[2];

[2]Atualmente não há suporte para variáveis do tipo **char**, **string** e **boolean**. Michael et al [MMS01] apresenta considerações importantes a respeito da dificuldade no tratamento destes tipos de dados, principalmente quando o tipo da variável é **string**.

4. Definir o domínio das variáveis de projeto, que especifica os limites superior e inferior para os valores de cada uma das variáveis;
5. Definir a precisão das variáveis de projeto;
6. Definir o valor do único parâmetro ajustável do GEO, τ;
7. Definir o limite superior para o número de avaliações da função objetivo *Similarity*;

A instrumentação do código-fonte é indispensável, pois somente com ela é possível determinar a cobertura alcançada por cada dado de teste. A instrumentação das arestas do CFG também é importante, uma vez que os caminhos possíveis do SUT são obtidos a partir dele. O tipo das variáveis de projeto é importante devido aos tipos exigidos por parâmetros de procedimentos e métodos. O domínio está intimamente relacionado ao número de bits necessários para a representação das variáveis de projeto no GEO, tal como a precisão das variáveis. É recomendável fazer um ajuste fino do parâmetro ajustável do GEO, o τ, visto que cada software pode ter uma característica que faz com que valores diferentes deste parâmetro tenham grande influência na eficiência da geração dos dados de teste. É importante comentar que esta observação vale também para outras metaheurísticas como, por exemplo, o AG.

6.2 A função objetivo *Similarity*

Como foi visto no Capítulo 4, a função objetivo *Similarity*, também conhecida como NEHD (em inglês, *Normalized Extended Hamming Distance*), foi proposta em 2001 por Lin e Yeh [LY01] e avalia a qualidade de soluções especificamente para o problema de teste orientado a caminhos. A qualidade de uma solução é avaliada através de uma série de equações e operações matemáticas de conjuntos. As entradas para a função objetivo são dois caminhos, sendo que um deles é definido como alvo. Um caminho alvo é um caminho do programa que deve obrigatoriamente ser coberto por um dado de teste.

As operações de conjunto são aplicadas aos caminhos, e as equações traduzem os resultados para um número, que indica a proximidade de um caminho para o caminho alvo. Observe que a proximidade do caminho alvo é diretamente proporcional à qualidade da solução (neste caso, o dado de teste). As vantagens e as limitações do uso desta função objetivo encontram-se no Capítulo 4, e o restante desta seção explicará *como* ela funciona.

A função objetivo *Similarity* foi escolhida por algumas razões. Primeiramente, ela é uma função objetivo desenvolvida especialmente para teste orientado a caminhos, o que vai de encontro com as necessidades deste trabalho. Em segundo lugar, apesar de os procedimentos internos da *Similarity* possuírem muitas operações de conjuntos e outros cálculos, a sua implementação é bem simples. A terceira razão é que ela trabalha em um nível de abstração que esconde a complexidade interna do SUT; por exemplo, não

importa se uma condição em uma decisão de um caminho é muito complexa ou não, ela somente considera que há uma decisão que deve ser coberta. Finalmente, esta função objetivo nunca foi avaliada para programas com *loops*. Apesar de seus autores, Lin e Yeh, afirmarem que a função objetivo funcionava para caminhos com *loops*, isto não foi avaliado. Os Capítulos 7 e 8 apresentam resultados que confirmam a habilidade desta função objetivo para avaliar caminhos deste tipo.

6.2.1 Conceitos importantes

Para um melhor entendimento da *Similarity*, alguns conceitos importantes serão apresentados inicialmente:

1. Conjunto de arestas: é um conjunto composto por arestas distintas de um determinado caminho, com elementos que variam de tamanho de acordo com a ordem do conjunto de arestas.
2. Tamanho do elemento de um conjunto de arestas: número de arestas combinadas em um único elemento do conjunto. Por exemplo, o elemento *ab* é de tamanho dois, *abc* de tamanho três, *abcd* de tamanho quatro, e assim sucessivamente.
3. Ordem de um conjunto de arestas: define o tamanho dos elementos do conjunto de arestas, ou seja, define o número de arestas que serão combinadas em um único elemento do conjunto. Por exemplo, suponha um caminho θ composto pelas arestas *a*, *b*, *c* e *d*, formando o caminho *abcd*; a Tabela 6.1 mostra os conjuntos de arestas para θ de ordem 1 até 4.

Tabela 6.1: Notação e conjuntos de arestas para o caminho θ.

Notação	Ordem	Conjunto de arestas
C^1_θ	1	a,b,c,d
C^2_θ	2	ab,bc,cd
C^3_θ	3	abc,bcd
C^4_θ	4	abcd

É importante ressaltar que a ordem dos conjuntos define o tamanho de cada elemento no conjunto de arestas, e que as arestas combinadas sempre estão imediatamente em seqüência. Isto descarta a possibilidade de uma combinação que origine um elemento *ac*, mesmo porque *ac* não é um fluxo de execução válido no programa, pois para se chegar à aresta *c* o fluxo de execução deve obrigatoriamente passar pela aresta *b*. Suponha agora um caminho β com *loops* composto pelas arestas *a*, *b* e *c*, formando o caminho *abbbc*; a Tabela 6.2 mostra os conjuntos de arestas para β de ordem 1 até 5.

Observe que a aresta *b* foi considerada somente uma única vez em C^1_β. Todo conjunto de arestas deverá considerar somente uma única vez arestas que se repetem

Tabela 6.2: Notação e conjuntos de arestas para o caminho β.

Notação	Ordem	Conjunto de arestas
C^1_β	1	a,b,b,b,c
C^2_β	2	ab,bb,bb,bc
C^3_β	3	abb,bbb,bbc
C^4_β	4	abbb,bbbc
C^5_β	5	abbbc

(estas indicam a presença de *loops* no caminho). O mesmo se aplica para C^2_β, onde um elemento repetido, *bb*, foi eliminado do conjunto. A quantidade de conjuntos de arestas para um caminho depende da ordem máxima para um conjunto de arestas. Cada caminho pode ter mais de um conjunto de arestas pois cada conjunto representa uma combinação diferente das arestas ao longo do caminho.

4. Ordem máxima para um conjunto de arestas: é o número de arestas distintas presentes no caminho alvo. Observe que *loops* percorrem mais de uma vez uma mesma aresta, logo as arestas envolvidas são consideradas apenas uma única vez neste cálculo, como se não houvesse *loops*. A notação utilizada será O^{MAX}_θ, sendo θ o caminho alvo. Como exemplo, suponha que o caminho β, formado pelas arestas *abbbc* (ver Tabela 6.2), seja o caminho alvo. A aresta *b* repete, logo será considerada apenas uma única vez. Sendo assim, $O^{MAX}_\beta = 3$.
5. Distância simétrica entre conjuntos de arestas: dados dois caminhos θ e β, e seus conjuntos de arestas de mesma ordem k, C^k_θ e C^k_β, a distância simétrica de ordem k entre estes conjuntos, chamada de $D^k_{\theta,\beta}$, é um conjunto formado por todos os elementos dos dois conjuntos (elementos repetidos devido à existência de *loops* são considerados apenas uma única vez), excluindo-se aqueles que são comuns aos dois. Matematicamente, $D^k_{\theta,\beta}$ equivale à Equação 6.1,

$$D^k_{\theta,\beta} = (C^k_\theta \cup C^k_\beta) - (C^k_\theta \cap C^k_\beta) = (\cup^k_{\theta,\beta} - \cap^k_{\theta,\beta}) \tag{6.1}$$

onde $\cup^k_{\theta,\beta}$ é o conjunto união dos elementos dos conjuntos de arestas de ordem k de θ e β, e $\cap^k_{\theta,\beta}$ o conjunto intersecção. Por exemplo, sejam os conjuntos de arestas de ordem 2 dos caminhos θ e β, C^2_θ e C^2_β, mostrados nas Tabelas 6.1 e 6.2, respectivamente. O conjunto $\cup^2_{\theta,\beta} = \{ab, bc, cd, bb\}$ e $\cap^2_{\theta,\beta} = \{ab, bc\}$. Sendo assim, $D^2_{\theta,\beta} = \{cd, bb\}$.
6. Distância normalizada entre conjuntos de arestas: é um número real no intervalo [0,1], resultante da normalização da distância simétrica entre conjuntos de arestas. Para o seu cálculo para uma determinada ordem k, basta dividir o número de elementos do conjunto da distância simétrica de ordem k pelo número de elementos do conjunto união de ordem k. Para obter o número de elementos de qualquer conjunto, basta aplicar nele uma operação de módulo; e.g. $|\{a, c, g\}| = 3$ (lê-se

módulo do conjunto a, c, g é igual a 3). A distância normalizada de ordem k entre os caminhos θ e β, chamada de $DN^k_{\theta,\beta}$, é definida pela Equação 6.2.

$$DN^k_{\theta,\beta} = \frac{\left|D^k_{\theta,\beta}\right|}{\left|\cup^k_{\theta,\beta}\right|} \tag{6.2}$$

Sabendo que $D^k_{\theta,\beta} = (\cup^k_{\theta,\beta} - \cap^k_{\theta,\beta})$ pela Equação 6.1, chega-se à Equação 6.3 por simplificação.

$$DN^k_{\theta,\beta} = \frac{\left|(\cup^k_{\theta,\beta} - \cap^k_{\theta,\beta})\right|}{\left|\cup^k_{\theta,\beta}\right|} = 1 - \frac{\left|\cap^k_{\theta,\beta}\right|}{\left|\cup^k_{\theta,\beta}\right|} \tag{6.3}$$

Observe que, caso $\left|\cap^k_{\theta,\beta}\right| = \left|\cup^k_{\theta,\beta}\right|$, $DN^k_{\theta,\beta} = 0$. Por outro lado, se $\left|\cap^k_{\theta,\beta}\right| = \oslash$, $DN^k_{\theta,\beta} = 1$. Isto garante que a distância normalizada sempre será um número real no intervalo [0,1].

Baseando-se no exemplo do item anterior para o cálculo da distância simétrica entre os caminhos θ e β, sejam os conjuntos $\cup^2_{\theta,\beta} = \{ab, bc, cd, bb\}$ e $\cap^2_{\theta,\beta} = \{ab, bc\}$. Logo, $DN^2_{\theta,\beta} = \frac{2}{4} = 0,5$.

7. Similaridade entre conjuntos de arestas: é um número real no intervalo [0,1] que indica a proximidade entre conjuntos de arestas de uma determinada ordem. A similaridade de ordem k entre os conjuntos de arestas de ordem k do caminho θ e β, chamada de $S^k_{\theta,\beta}$, é dada pela Equação 6.4.

$$S^k_{\theta,\beta} = \frac{\left|\cap^k_{\theta,\beta}\right|}{\left|\cup^k_{\theta,\beta}\right|} = 1 - DN^k_{\theta,\beta} \tag{6.4}$$

Note que caso $S^k_{\theta,\beta} = 0$, não há nenhum elemento comum entre os conjuntos de arestas de ordem k dos caminhos θ e β, ou seja, $\cap^k_{\theta,\beta} = \oslash$. Por outro lado, se $S^k_{\theta,\beta} = 1$, todos os elementos dos conjuntos são comuns entre θ e β. O caminho β somente será idêntico ao caminho θ caso $S^k_{\theta,\beta} = 1 \quad \forall \quad k = 1, ..., O^{MAX}_\theta$, supondo que θ é o caminho alvo. Sabendo o valor da distância normalizada para qualquer ordem, o cálculo da similaridade de qualquer ordem é trivial. Por exemplo, voltando ao exemplo do item anterior (cálculo da distância normalizada), $S^2_{\theta,\beta} = 1 - DN^2_{\theta,\beta} = 1 - 0,5 = 0,5$.

8. Fatores de peso: são multiplicadores aplicados às similaridades entre conjuntos de arestas. Quanto maior a ordem da similaridade entre os conjuntos, maior o fator de peso aplicado a ela. Os fatores de peso são calculados a partir do número de elementos do conjunto de arestas de cada ordem do caminho alvo, além do fator de peso anterior. A Equação 6.5 define o cálculo do fator de peso de ordem k com base

no caminho alvo θ, chamado de P_θ^k.

$$P_\theta^k = \left|C_\theta^{k-1}\right| * P_\theta^{k-1}, \tag{6.5}$$

sendo que $P_\theta^1 = 1, k = 2, ..., O_\theta^{MAX}$. Dessa forma, fica claro que o fator de peso P_θ^{k+1} será $\left|C_\theta^k\right|$ vezes maior que P_θ^k.

Suponha que o caminho alvo seja o caminho β da Tabela 6.2. Vimos que a ordem máxima para β, O_β^{MAX}, é 3. Sendo assim, o maior fator de peso será P_β^3. Os cálculos encontram-se na Tabela 6.3 logo abaixo.

Tabela 6.3: Fatores de peso baseando-se no caminho alvo β.

k	Cálculo	P^k
1	1, por definição	1
2	$\left\|C_\beta^1\right\| * P_\beta^1 = 3 * 1$	3
3	$\left\|C_\beta^2\right\| * P_\beta^2 = 3 * 3$	9

9. Similaridade total entre caminhos: é um número real que corresponde à soma das similaridades de todas as ordens entre dois caminhos, sendo que cada uma das similaridades é multiplicada por um fator de peso específico, de acordo com a ordem da similaridade. Quanto maior o seu valor, maior a proximidade de um caminho para o caminho alvo. A similaridade total entre dois caminhos θ e β (sendo que θ é o caminho alvo), chamada de $SIM_{\theta,\beta}$, é calculada de acordo com a Equação 6.6.

$$SIM_{\theta,\beta} = S^1 * P_\theta^1 + S^2 * P_\theta^2 + \ldots + S^{O^{MAX}} * P_\theta^{O^{MAX}} \tag{6.6}$$

6.2.2 Funcionamento da *Similarity*

A *Similarity* não leva em consideração o resultado das decisões ao longo dos caminhos de um SUT, mas somente as arestas entre as decisões. Partindo de dois caminhos, sendo um deles um caminho alvo, ela inicia seus cálculos obtendo a ordem máxima para o conjunto de arestas. Como foi visto na seção anterior, esta ordem equivale ao número de arestas distintas do caminho alvo.

A partir deste momento, são feitos os cálculos dos conjuntos de arestas, distância simétrica, distância normalizada e similaridade entre os dois caminhos, nesta ordem. Os cálculos sempre começam com a ordem 1 e só são interrompidos após os cálculos para a ordem máxima **ou** quando a similaridade de qualquer ordem for igual a zero ou a um (nesta última situação, o caminho alvo é coberto), sendo este o critério de parada no fluxograma da Figura 6.2.

Após isto, os fatores de peso são calculados e, de posse das similaridades de todas as ordens entre os dois caminhos, é possível calcular a similaridade total entre eles. Note

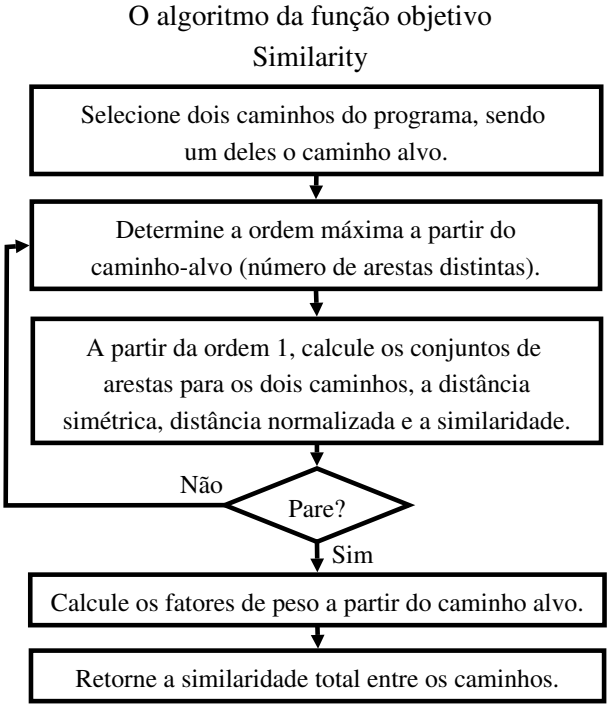

Figura 6.2: Fluxograma da função objetivo *Similarity*.

que a referência principal é sempre o caminho alvo, tanto que o cálculo dos fatores de peso e ordem máxima utilizam informações inerentes a ele. A saída do algoritmo será um número real indicando a proximidade entre os dois caminhos, sendo que quanto maior o valor, maior a proximidade do caminho avaliado para o caminho alvo.

6.2.3 Um exemplo prático utilizando a *Similarity*

Para demonstrar o funcionamento da *Similarity*, será dado um exemplo prático utilizando um programa clássico para o cálculo do número de Fibonacci, cujo código-fonte e CFG instrumentado estão apresentados na Figura 6.3.

Este programa possui dez blocos básicos, sendo que os blocos 0 e 9 representam o bloco de entrada e saída, respectivamente. Como foi visto no Capítulo 2, um caminho completo em um programa deve obrigatoriamente começar em um bloco de entrada e terminar em um de saída. Este requisito nos permite extrair inúmeros caminhos do programa da Figura 6.3. Note que cada uma das arestas do CFG foram marcadas (ou instrumentadas) com letras; estas instrumentações no CFG equivalem a instrumentações inseridas no código-fonte do programa, que podem ser feitas com uma instrução *printf(...)* ou *System.out.println(...)*, por exemplo. Dessa forma, um caminho é formado por uma seqüência de caracteres (que seriam as arestas), começando no bloco básico 0 e encerrando no bloco

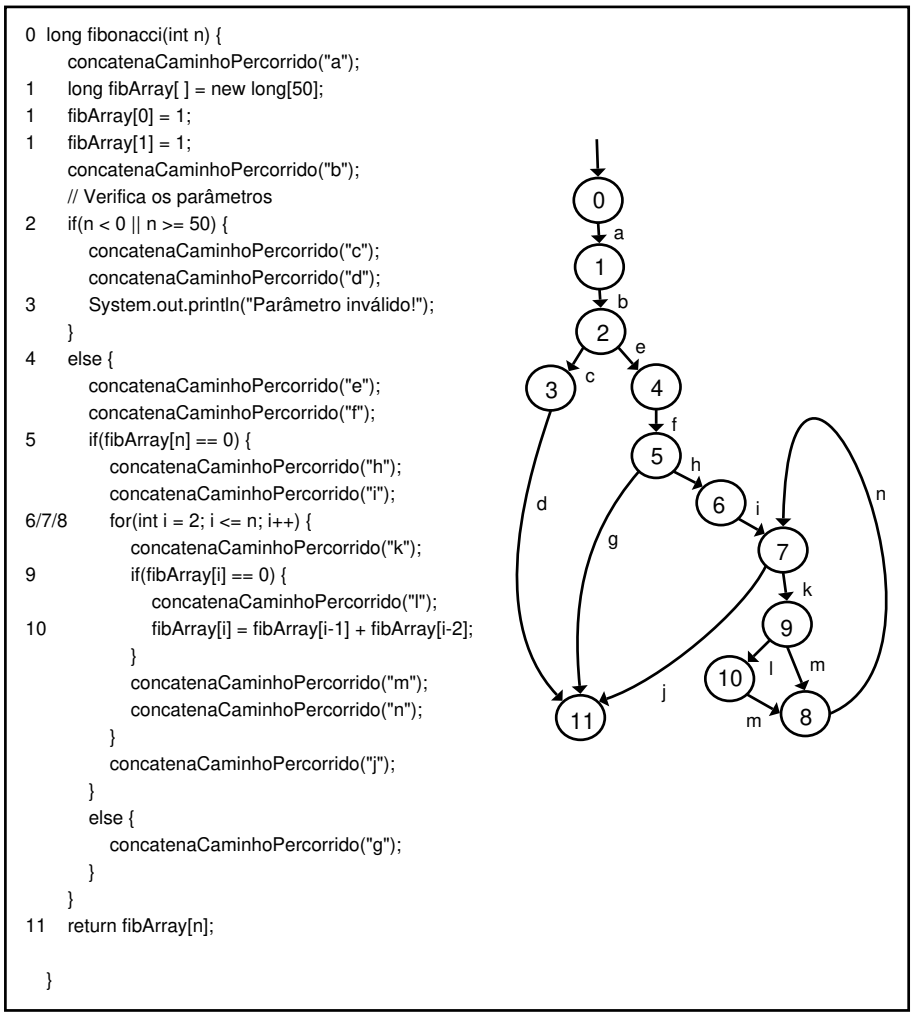

Figura 6.3: Código-fonte e CFG instrumentados do programa Fibonacci.

básico 9. A Tabela 6.4 a seguir apresenta um subconjunto de caminhos selecionados aleatoriamente a partir do CFG na Figura 6.3. A seqüência de caracteres envolvidas em $\langle \ldots \rangle$ e em itálico indicam *loops*.

Suponha que o caminho 4 (CA4) seja selecionado como caminho alvo. Os caminhos que serão comparados com ele serão os caminhos 3 (CA3) e 9 (CA9), nesta ordem. O CA3 e CA9 foram escolhidos para demonstrar o funcionamento da função objetivo para caminhos com *loops* (CA9) e sem *loops* (CA3). Seguindo os passos do fluxograma da Figura 6.2, a ordem máxima é calculada. Sabendo que o número de arestas distintas do caminho alvo CA4 é 10, a ordem máxima $O_{CA4}^{MAX} = 10$. A partir daí, os conjuntos de arestas para os dois caminhos, distâncias simétricas, distâncias normalizadas e as similaridades são calculadas, iniciando-se na ordem $k = 1$ e indo até $k = O_{CA4}^{MAX}$. O processo é interrompido

Tabela 6.4: Subconjunto de caminhos.

Caminho	Seqüência de caracteres
1	abcd
2	abefg
3	abefhij
4	abefhi$\langle kmn\rangle$j
5	abefhi$\langle kmn\rangle$ $\langle kmn\rangle$j
6	abefhi$\langle kmn\rangle$ $\langle kmn\rangle$ $\langle kmn\rangle$j
7	abefhi$\langle klmn\rangle$j
8	abefhi$\langle klmn\rangle$ $\langle klmn\rangle$j
9	abefhi$\langle klmn\rangle$ $\langle kmn\rangle$ $\langle klmn\rangle$j

somente se $S^k_{CA4,CA3} = 0$ ou $S^k_{CA4,CA3} = 1$ durante os cálculos. Este último caso implica na cobertura do caminho alvo, o que justifica a interrupção nos cálculos. A Tabela 6.5 apresenta os cálculos passo-a-passo para CA4 e CA3.

Tabela 6.5: Cálculo das similaridades $S^k_{CA4,CA3}$.

k	Caminho	C^k	$\cup^k$	$\cap^k$	D^k	DN^k	S^k
1	CA4	a,b,e,f,h,i,k,m,n,j	a,b,e,f,h,i, k,m,n,j	a,b,e,f,h,i,j	k,m,n	$\frac{3}{10}$	0,70
	CA3	a,b,e,f,h,i,j					
2	CA4	ab,be,ef,fh,hi, i*k*,*km*,*mn*,*nj*[a]	ab,be,ef,fh, hi,ik,km,mn, nj,ij	ab,be,ef,fh,hi	ik,km,mn,nj, ij	$\frac{5}{10}$	0,50
	CA3	ab,be,ef,fh,hi,ij					
3	CA4	abe,bef,efh,fhi, hi*k*,i*km*,*kmn*,*mn*j	abe,bef,efh, fhi,hik,ikm, kmn,mnj,hij	abe,bef,efh, fhi	hik,ikm,kmn, mnj,hij	$\frac{5}{9}$	0,44
	CA3	abe,bef,efh,fhi,hij					
4	CA4	abef,befh,efhi, fhi*k*,hi*km*,i*kmn*, *kmn*j	abef,befh,efhi, fhik,hikm,ikmn, kmnj,fhij	abef,befh, efhi	fhik,hikm, ikmn,kmnj, fhij	$\frac{5}{8}$	0,38
	CA3	abef,befh,efhi,fhij					
5	CA4	abefh,befhi,efhi*k*, fhi*km*,hi*kmn*,i*kmn*j	abefh,befhi,efhik, fhikm,hikmn, ikmnj,efhij	abefh,befhi	efhik,fhikm, hikmn,ikmnj, efhij	$\frac{5}{7}$	0,29
	CA3	abefh,befhi,efhij					
6	CA4	abefhi,befhi*k*, efhi*km*,fhi*kmn*, hi*kmn*j	abefhi,befhik, efhikm,fhikmn, hikmnj,befhij	abefhi	befhik,efhikm, fhikmn,hikmnj, befhij	$\frac{5}{6}$	0,17
	CA3	abefhi,befhij					
7	CA4	abefhi*k*,befhi*km*, efhi*kmn*,fhi*kmn*j	abefhik,befhikm, efhikmn,fhikmnj, abefhij	$\oslash$	abefhik,befhikm, efhikmn,fhikmnj, abefhij	$\frac{5}{5}$	0,00[b]
	CA3	abefhij					

[a]Note que os *loops* já são considerados (arestas destacadas em itálico).
[b]Como $S^7 = 0$, o algoritmo pára neste momento.

É importante ressaltar que os valores de $S^k_{CA4,CA3}$ para $k = 8, 9, 10$ serão iguais a zero, o que dispensa o seu cálculo. Os cálculo dos fatores de peso P^k_{CA4} são mostrados na Tabela 6.6.

De posse dos valores dos fatores de peso, a similaridade total entre CA4 e CA3 será

Tabela 6.6: Fatores de peso.

k	Cálculo	P^k
1	1, por definição	1
2	$\|C^1_{CA4}\| * P^1_{CA4} = 10 * 1$	10
3	$\|C^2_{CA4}\| * P^2_{CA4} = 9 * 10$	90
4	$\|C^3_{CA4}\| * P^3_{CA4} = 8 * 90$	720
5	$\|C^4_{CA4}\| * P^4_{CA4} = 7 * 720$	5040
6	$\|C^5_{CA4}\| * P^5_{CA4} = 6 * 5040$	30240
7	$\|C^6_{CA4}\| * P^6_{CA4} = 5 * 30240$	151200
8	$\|C^7_{CA4}\| * P^7_{CA4} = 4 * 151200$	604800
9	$\|C^8_{CA4}\| * P^8_{CA4} = 3 * 604800$	1814400
10	$\|C^9_{CA4}\| * P^9_{CA4} = 2 * 1814400$	3628800

dada pela Equação 6.7 abaixo.

$$SIM_{CA4,CA3} = S^1 * P^1_{CA4} + S^2 * P^2_{CA4} + \ldots + S^{O^{MAX}} * P^{O^{MAX}}_{CA4} \tag{6.7}$$

$$SIM_{CA4,CA3} = 0,70*1+0,50*10+0,44*90+0,38*720+0,29*5040+0,17*30240+0*151200$$

$$SIM_{CA4,CA3} = 0,70 + 5,00 + 39,60 + 273,60 + 1461,60 + 5140,80 + 0$$

$$SIM_{CA4,CA3} = 6921,3$$

A partir de agora, os mesmos cálculos serão feitos para comparar CA4 e CA9 e boa parte dos cálculos feitos para o caminho alvo, CA4, serão reaproveitados. Porém, como o caminho CA9 possui um número maior de arestas, inclusive *loops*, os resultados intermediários e final certamente serão diferentes, como pode ser visto nos cálculos passo-a-passo da Tabela 6.7 no final deste capítulo.

Os fatores de peso estão diretamente associados ao número de elementos dos conjuntos de arestas do caminho alvo[3]. Como o caminho alvo é o mesmo, os mesmos fatores de peso apresentados na Tabela 6.6 poderão ser utilizados. A similaridade total entre CA4 CA9 será, então,

$$SIM_{CA4,CA9} = S^1 * P^1_{CA4} + S^2 * P^2_{CA4} + \ldots + S^{O^{MAX}} * P^{O^{MAX}}_{CA4}$$

$$SIM_{CA4,CA9} = 0,91 * 1 + 0,75 * 10 + 0,57 * 90 + 0,24 * 720 + 0,18 * 5040 +$$

$$0,13 * 30240 + 0,07 * 151200 + 0 * 604800$$

$$SIM_{CA4,CA9} = 0,91 + 7,50 + 51,30 + 172,80 + 907,20 + 3931,20 + 10584,00 + 0$$

$$SIM_{CA4,CA9} = 15654,91$$

[3]Vale lembrar que o primeiro fator de peso é sempre igual a 1, e os demais são calculados utilizando o número de elementos dos conjuntos do caminho alvo, como mostrou a Equação 6.5.

Observe que $SIM_{CA4,CA9} > SIM_{CA4,CA3}$, o que significa que o caminho 9 está mais próximo do caminho alvo 4 do que o caminho 3. Sabendo estes valores, é possível classificar cada caminho com relação a outro e, conseqüentemente, esta informação será utilizada para direcionar a busca por soluções que se aproximem o máximo possível do caminho definido como alvo.

6.3 Procedimento de geração dos dados de teste

A Figura 6.4 apresenta uma descrição alto-nível do procedimento de geração dos dados para teste de caminhos. Dado um conjunto de caminhos-alvo, um deles é selecionado e, a partir deste momento, o GEO começa a gerar dados de teste tentando cobri-lo. Cada dado gerado é uma entrada para o SUT, que por sua vez é executado com este dado. Graças à instrumentação previamente inserida no código-fonte, é possível saber qual foi o caminho coberto pelo dado de teste. O caminho coberto é a saída do SUT, e será entrada para a função objetivo *Similarity*, que calculará a proximidade entre o caminho coberto e o caminho alvo. Se o caminho coberto for igual a qualquer caminho alvo presente no conjunto inicial, este caminho é removido do conjunto e o dado de teste gerado é armazenado, associado ao caminho alvo que foi coberto. Este fenômeno é conhecido como cobertura colateral (em inglês, *serendipitous coverage* [MMS01]), uma vez que a busca por um dado para cobrir um determinado requisito cobriu outros requisitos.

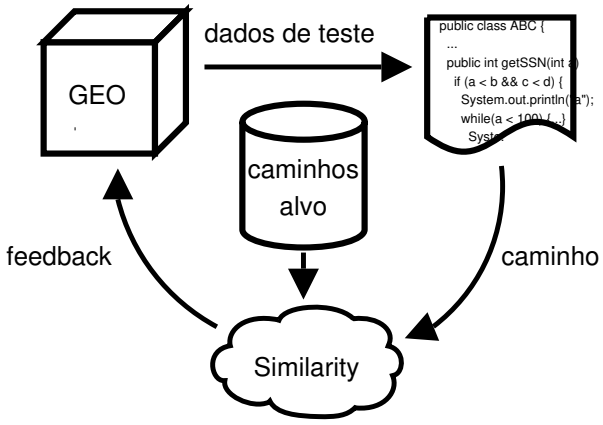

Figura 6.4: A abordagem dinâmica utilizada.

O GEO pára e reinicia o processo de geração de dados utilizando outro caminho como alvo somente quando o caminho coberto for igual ao alvo. Caso contrário, ele prosseguirá com a geração até que algum critério de parada seja satisfeito. Atualmente, o processo é interrompido quando o número máximo de avaliações da função objetivo *Similarity* é alcançado, ou quando todos os caminhos presentes no conjunto inicial de caminhos-alvo

forem cobertos. Vale lembrar que o valor de saída da *Similarity* é o retorno da qualidade do dado de teste, que é entrada para o GEO de forma a auxiliá-lo a melhorar a qualidade dos dados de teste[4], aproximando-os cada vez mais de um dado que efetivamente cubra o caminho alvo.

Também deve ser ressaltado que outros critérios de adequabilidade poderiam ser utilizados neste trabalho. A função objetivo é geralmente associada a um critério de adequabilidade; logo, para o teste de ramos (ou arestas), seria necessário trocar a *Similarity* por uma função objetivo apropriada para teste de ramos. A instrumentação do SUT poderia ser modificada com o objetivo de coletar outras informações a respeito da execução do software. Analogamente ao teste de caminhos, existiria um conjunto inicial de ramos-alvo que precisariam ser cobertos, tal como o conjunto de caminhos-alvo mostrado na Figura 6.4.

[4]Vale lembrar que a qualidade do dado de teste é um valor que indica a proximidade do caminho coberto pelo dado de teste do caminho alvo; quanto mais alto este valor, mais próximo o caminho coberto está do caminho alvo.

Tabela 6.7: Cálculo das similaridades $S^k_{CA4,CA9}$.

k	Caminho	C^k	$\cup^k$	$\cap^k$	D^k	DN^k	S^k
1	CA4	a,b,e,f,h,i,k,m,n,j	a,b,e,f,h,i,k,m,n,j,l	a,b,e,f,h, i,k,m,n,j	e	$\frac{1}{11}$	0,91
	CA9	a,b,e,f,h,i,k,l,m,n,j					
2	CA4	ab,be,ef,fh,hi, i*k*,*km*,*mn*,*nj*[a]	ab,be,ef,fh,hi,ik, km,mn,nj,kl,lm,nk	ab,be,ef, fh,hi,ik, km,mn, nj	kl,lm,nk lj,mi	$\frac{3}{12}$	0,75
	CA9	ab,be,ef,fh,hi,i*k*, *kl*,*lm*,*mn*,*nk*,*km*, *mn*,*nk*,*kl*,*lm*,*mn*,nj					
3	CA4	abe,bef,efh,fhi, hi*k*,i*km*,*kmn*,*mnj*	abe,bef,efh,fhi, hik,ikm,kmn,mnj, ikl,klm,lmn,mnk, nkm,nkl	abe,bef, efh,fhi, hik,kmn, mnj	ikl,klm,lmn, mnk,nkm,nkl	$\frac{6}{14}$	0,57
	CA9	abe,bef,efh,fhi, hi*k*,i*kl*,*klm*,*lmn*, *mnk*,*nkm*,*kmn*,*mnk*, *nkl*,*klm*,*lmn*,*mnj*					
4	CA4	abef,befh,efhi,fhi*k*, hi*km*,i*kmn*,*kmnj*	abef,befh,efhi, fhik,hikm,ikmn, kmnj,hikl,iklm, klmn,lmnk,mnkm, nkmn,kmnk,mnkl, nklm,lmnj	abef,befh, efhi,fhik	hikm,ikmn, kmnj,hikl, iklm,klmn, lmnk,mnkm, nkmn,kmnk, mnkl,nklm, lmnj	$\frac{13}{17}$	0,24
	CA9	abef,befh,efhi,fhi*k*, hi*kl*,i*klm*,*klmn*, *lmnk*,*mnkm*,*nkmn*, *kmnk*,*mnkl*,*nklm*, *klmn*,*lmnj*					
5	CA4	abefh,befhi,efhi*k*, fhi*km*,hi*kmn*,i*kmnj*	abefh,befhi,efhik, fhikm,hikmn,ikmnj, fhikl,hiklm,iklmn, klmnk,lmnkm, mnkmn,nkmnk, kmnkl,mnklm, nklmn,klmnj	abefh, befhi, efhik	fhikm,hikmn, ikmnj,fhikl, hiklm,iklmn, klmnk,lmnkm, mnkmn,nkmnk, kmnkl,mnklm, nklmn,klmnj	$\frac{14}{17}$	0,18
	CA9	abefh,befhi,efhi*k*, fhi*kl*,hi*klm*,i*klmn*, *klmnk*,*lmnkm*,*mnkmn* *nkmnk*,*kmnkl*,*mnklm* *nklmn*,*klmnj*,					
6	CA4	abefhi,befhi*k*, efhi*km*,fhi*kmn*, hi*kmnj*	abefhi,befhik, efhikm,fhikmn, hikmnj,efhikl, fhiklm,hiklmn, iklmnk,klmnkm, lmnkmn,mnkmnk, nkmnkl,kmnklm, mnklmn,nklmnj	abefhi, befhik	efhikm,fhikmn, hikmnj,efhikl, fhiklm,hiklmn, iklmnk,klmnkm, lmnkmn,mnkmnk, nkmnkl,kmnklm, mnklmn,nklmnj	$\frac{14}{16}$	0,13
	CA9	abefhi,befhi*k*,efhi*kl*, fhi*klm*,hi*klmn*,i*klmnk*, *klmnkm*,*lmnkmn*, *mnkmnk*,*nkmnkl*, *kmnklm*,*mnklmn*, *nklmnj*					
7	CA4	abefhi*k*,befhi*km*, efhi*kmn*,fhi*kmnj*	abefhik,befhikm, efhikmn,fhikmnj, befhikl,efhiklm, fhiklmn,hiklmnk, iklmnkm,klmnkmn, lmnkmnk,mnkmnkl, nkmnklm,kmnklmn, mnklmnj	abefhik	befhikm,efhikmn, fhikmnj,befhikl, efhiklm,fhiklmn, hiklmnk,iklmnkm, klmnkmn,lmnkmnk, mnkmnkl,nkmnklm, kmnklmn,mnklmnj	$\frac{14}{15}$	0,07
	CA9	abefhi*k*,befhi*kl*,efhi*klm*, fhi*klmn*,hi*klmnk*, i*klmnkm*,*klmnkmn*, *lmnkmnk*,*mnkmnkl*, *nkmnklm*,*kmnklmn*, *mnklmnj*					
8	CA4	abefhi*km*,befhi*kmn*, efhi*kmnj*	abefhikm,befhikmn, efhikmnj,abefhikl, befhiklm,efhiklmn, fhiklmnk,hiklmnkm, iklmnkmn,klmnkmnk, lmnkmnkl,mnkmnklm, nkmnklmn,kmnklmnj	⊘	abefhikm,befhikmn, efhikmnj,abefhikl, befhiklm,efhiklmn, fhiklmnk,hiklmnkm, iklmnkmn,klmnkmnk, lmnkmnkl,mnkmnklm, nkmnklmn,kmnklmnj	$\frac{14}{14}$	0,00[b]
	CA9	abefhi*kl*,befhi*klm*, efhi*klmn*,fhi*klmnk*, hi*klmnkm*,i*klmnkmn*, *klmnkmnk*,*lmnkmnkl*, *mnkmnklm*, *nkmnklmn*,*kmnklmnj*					

[a]Note que os *loops* já são considerados (arestas destacadas em itálico).

[b]Como $S^8 = 0$, o algoritmo pára neste momento.

Capítulo 7

Resultados utilizando estudos de caso simples

Os resultados apresentados neste capítulo são uma evolução de um primeiro trabalho publicado por Abreu et al [AMS05] no Simpósio Brasileiro de Engenharia de Software (SBES), em 2005. Neste primeiro trabalho, o GEO foi aplicado pela primeira vez em um problema de Engenharia de Software. O GEO foi utilizado em uma abordagem dinâmica para a geração de dados de teste para a cobertura de caminhos, sendo avaliado inicialmente utilizando o programa do triângulo simplificado como estudo de caso. Nesta publicação, o GEO foi comparado com o SGA e *Random-Test*, que gera dados de teste de forma aleatória. A Seção 7.1 a seguir apresenta um dos resultados obtidos nesta publicação, enquanto que a Seção 7.2 descreve a avaliação do GEO para estudos de caso simples.

7.1 Resultados obtidos anteriormente

Dentre os resultados obtidos por Abreu et al., um dos mais interessantes foi uma comparação entre a evolução do melhor valor médio de aptidão para o GEO e o SGA, mostrado na Figura 7.1, retirada de [AMS05]. O melhor valor médio de aptidão utilizando o SGA aumentou rapidamente durante as primeiras 32000 avaliações da *Similarity* e alcançou o valor de 1,5, visto que praticamente todas as execuções até aquele ponto já tinham encontrado dados de teste para cobrir o caminho do triângulo isósceles, cuja aptidão associada era de 1,5. Depois disso, dados para cobrir o caminho do triângulo equilátero começaram a ser encontrados e contribuíram para aumentar o valor médio de aptidão até a marca de 3,03, com 100000 avaliações da *Similarity*. Note na curva do SGA que a maioria das execuções com sucesso ocorreram no intervalo de 32000 a 100000 avaliações da função objetivo, uma vez que a melhor média de aptidão ultrapassou 1,5 a partir deste número de avaliações.

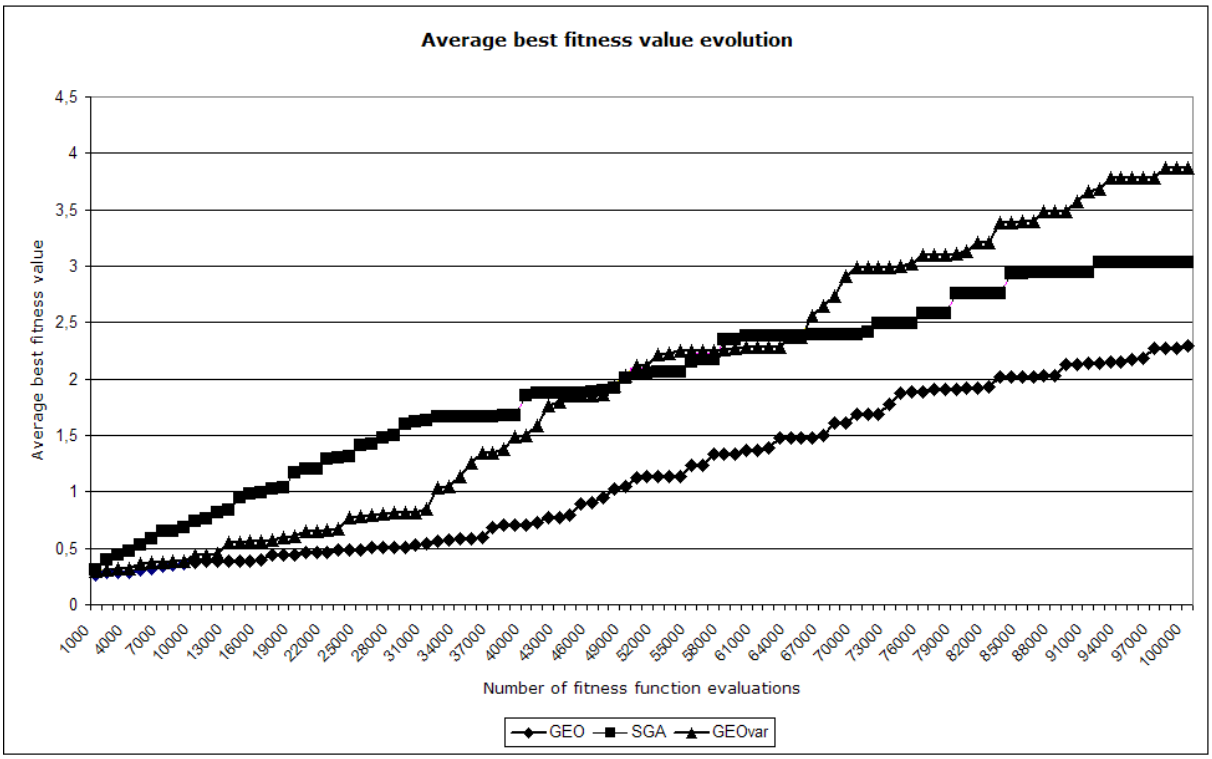

Figura 7.1: Evolução do melhor valor médio de aptidão utilizando o GEO e o SGA.

Por outro lado, apesar do melhor valor médio de aptidão utilizando o GEO_{var} ter evoluído vagarosamente em um primeiro momento, ele obteve um número maior de execuções com sucesso, ou seja, encontrou mais vezes dados de teste para cobrir o caminho do triângulo equilátero. A maioria destas execuções com sucesso também ocorreram durante a mesma faixa de avaliações da *Similarity* para o SGA, mas a taxa de crescimento do GEO_{var} foi claramente superior que a do SGA. Esta taxa maior explica o fato de o GEO_{var} ter obtido um desempenho melhor que o SGA para encontrar dados para o triângulo equilátero. Os outros resultados obtidos encontram-se em [AMS05].

7.2 Avaliando o GEO em estudos de casos simples

Neste capítulo, o desempenho do GEO foi analisado em sete programas na linguagem Java, chamados de programas de estudo (em inglês, *Subject Programs*, SPs), que também foram utilizados por outros autores em seus trabalhos [MS04, MMS01, PHP99, Sth96]: triângulo simplificado (ou *simplified triangle*), resto (ou *remainder*), produto (ou *product*), busca linear (ou *linear search*), busca binária (ou *binary search*), valor do meio (ou *middle value*) e triângulo (ou *triangle*). O primeiro SP classifica um triângulo em quatro categorias

(equilátero, isósceles, escaleno ou "não é um triângulo"), enquanto a segunda e terceira calculam o resto da divisão e o produto de dois valores, respectivamente. A quarta e a quinta realizam uma busca linear e binária de uma chave fornecida pelo usuário em um vetor, respectivamente. A sexta encontra o número do meio, dados três números, enquanto que o último SP classifica um triângulo em seis categorias (equilátero, isósceles, reto, acutângulo, obtusângulo ou "não é um triângulo"). Todos os experimentos foram executados em um Intel XEON CPU com 2.40GHz, quatro processadores e 1GB de RAM.

Cada um destes SPs foi instrumentado manualmente através da inserção de instruções *concatenaCaminhoPercorrido(...)*[1]. Dando seqüência, os CFGs foram desenhados manualmente, um para cada SP, sendo que cada aresta foi nomeada de acordo com a instrumentação inserida no código do SP. De posse dos CFGs com as arestas nomeadas, foi possível extrair um subconjunto de caminhos a serem cobertos de cada SP, chamados de caminhos-alvo. A Figura 7.2 a seguir mostra um exemplo de código-fonte com a instrumentação já inserida e seu CFG correspondente.

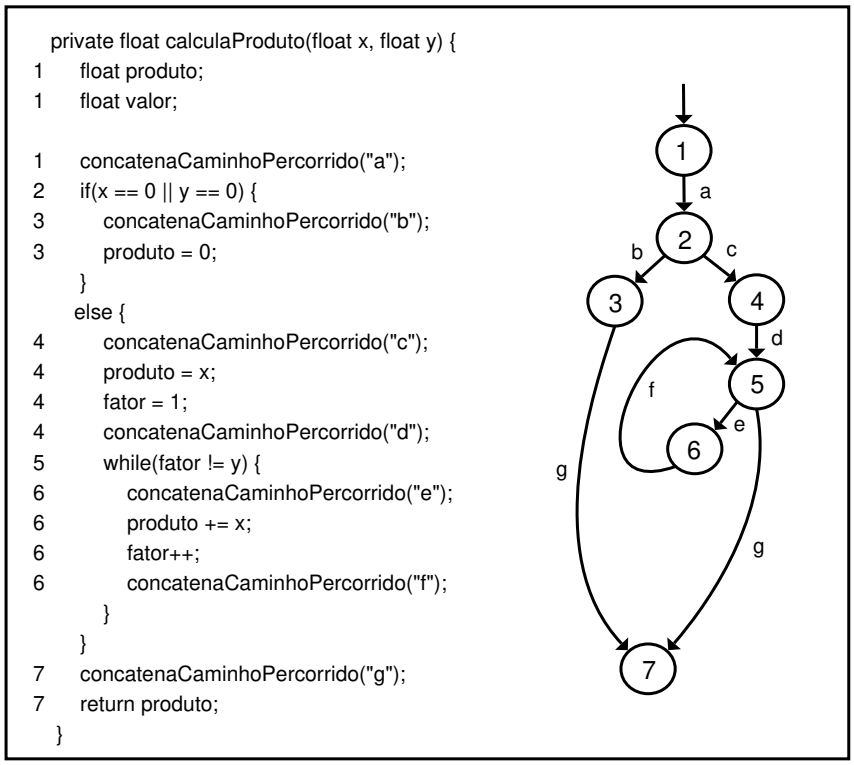

Figura 7.2: Exemplo de código-fonte instrumentado e seu CFG.

Estes SPs possuem complexidade e tipos de parâmetros diferentes, como mostra a

[1]Esta instrução recebe um caracter como parâmetro e armazena-o no final de um *string* que representa o caminho percorrido até o momento pelo programa.

Tabela 7.1: Características dos SPs.

#	SP	NIV[a]	Tipo	Domínio	NTP[b]	CC[c]
1	triângulo simplificado	3	inteiro	[0,65535]	4	13
2	resto	2	inteiro	[0,65535]	5	2
3	produto	2	inteiro	[0,1023]	6	4
4	busca linear	1	inteiro	[0,16383]	5	4
5	busca binária	1	inteiro	[0,16383]	12	5
6	valor do meio	3	inteiro	[-32768,32767]	4	6
7	triângulo	3	inteiro	[0,65535]	6	10

[a]Número de variáveis de entrada.
[b]Número de caminhos-alvo.
[c]Complexidade ciclomática de McCabe [McC76], medida utilizando a ferramenta Metrics [Met].

Tabela 7.1. Os SPs de 2 até 5 possuem caminhos-alvo contendo *loops*, sendo que estes requerem que cada *loop* seja executado zero, uma, duas e mais de duas vezes. A abordagem para tratamento dos *loops* neste trabalho é a mesma utilizada por Sthamer [Sth96] em sua tese. O domínio das variáveis de entrada foram definidos de forma a tornar mais difícil a busca por soluções viáveis, visto que domínios pequenos reduzem o espaço de projeto do problema, conseqüentemente aumentando a chance de sucesso de uma abordagem completamente aleatória, por exemplo.

O GEO foi comparado com o SGA e *Random-Test* utilizando como critério a percentagem média de cobertura adquirida durante a geração dos dados de teste[2] e o tempo consumido na geração dos dados de teste. A percentagem de cobertura está associada à métrica de software chamada de cobertura de código. Como será visto mais tarde neste capítulo, a análise deste critério encapsula outros critérios importantes, como o número de dados de teste gerados antes de atingir uma determinada percentagem de cobertura. Neste trabalho, o tipo de cobertura de código analisada é a de caminhos, ou seja, dado um conjunto de caminhos a serem testados, qual a percentagem de caminhos efetivamente cobertos ao final do processo.

É importante lembrar que nos SPs 4 e 5 somente a chave de busca será gerada e evoluída durante cada iteração do processo de busca, enquanto que os elementos do vetor são inicializados aleatoriamente no início de cada processo. Os tamanhos de vetor utilizados para o SP 4 e SP 5 foram 13 e 40, respectivamente. Apesar de todas os SPs não considerarem outros tipos de variável como, por exemplo, *string* e *boolean*, a abordagem utilizada pode lidar com estes tipos de dados. Para que isto seja possível, o projetista de testes incluiria um passo de conversão, responsável por converter o valor produzido pelo GEO em um caracter ou valor booleano. No caso de uma variável do tipo *boolean*, o GEO trabalharia com uma variável do tipo inteiro, com um domínio [0,1]. Sendo assim, o valor 1 seria convertido para **true** e 0, para **false**.

[2]Um dado de teste gerado também corresponde a uma avaliação da função objetivo, resultando em uma execução do SUT.

No caso de variáveis do tipo *string* a informação útil a ser utilizada é que um *string* é um vetor de caracteres. Dessa forma, uma variável do tipo *string* com um número máximo de caracteres limitados a trinta[3] seria representada no GEO como sendo trinta variáveis de projeto, cada uma do tipo inteiro no domínio [0,255], no intuito de representar todos os caracteres do tipo ASCII. Após a geração dos valores, cada um deles seria convertido para seu caracter ASCII equivalente. No entanto, a geração de *strings* é um problema quando eles precisam satisfazer determinadas restrições de forma que a entrada faça sentido (por exemplo, um nome de arquivo que precisa ser aberto e lido). Além disso, estas restrições podem variar de um SUT para outro, fazendo com que uma automação com suporte universal para este caso seja bastante difícil [MMS01].

7.2.1 Ajustando os parâmetros dos algoritmos

O desempenho do GEO e do SGA pode variar de forma significativa com os valores de seus parâmetros ajustáveis, logo eles foram devidamente ajustados para cada um dos SPs. É importante comentar que o *Random-Test* não necessita de ajustes em virtude do mesmo não possuir parâmetros ajustáveis. Por outro lado, o GEO possui um único parâmetro, τ, enquanto que o SGA possui três: probabilidades de *crossover* (p_c) e mutação (p_m), além do tamanho da população (*popsize*). O ajuste foi feito aplicando o GEO e o SGA com diversas combinações de parâmetros em cada SP da Tabela 7.1. As demais condições para o ajuste foram:

- Cada algoritmo foi aplicado 100 vezes para cada combinação de parâmetros possível.
- O número máximo de avaliações da função objetivo *Similarity* foi limitado em 100000.
- Tanto o GEO quanto sua variação, GEO_{var}, iniciaram a busca a partir da mesma solução inicial, gerada aleatoriamente.
- O parâmetro τ (GEO e GEO_{var}) variou de 0 a 10, com incrementos de 0,25.
- Para o SGA:

$$\begin{cases} 100 \leq popsize \leq 10000 & \text{começando em 100 e com incrementos de } popsize * 10, \\ 0,6 \leq p_c \leq 1,0 & \text{começando em 0,6 e com incrementos de 0,1,} \\ 0,0010 \leq p_m \leq 0,0205 & \text{começando em 0,0010 e com incrementos de 0,0015} \end{cases}$$

Os incrementos para os parâmetros de cada algoritmo foram definidos de forma a tornar o ajuste, tanto do GEO quanto do SGA, o mais preciso possível.

[3]Considerando que esta informação a respeito de limites deve constar no documento de especificação do sistema.

Está claro que o GEO possui menos combinações de parâmetros que o SGA, o que torna seu processo de ajuste muito menos dispendioso computacionalmente que o do SGA. Por exemplo, de acordo com as condições de ajuste dos algoritmos especificadas anteriormente, o GEO possui 41 combinações possíveis de parâmetros enquanto o SGA possui 210 (3∗5∗14), ou seja, aproximadamente 5 vezes mais combinações que o GEO. Os resultados do ajuste apresentados na Tabela 7.2 mostram combinações diferentes de parâmetros do SGA para cada SP, confirmando que o ajuste dos parâmetros realmente é importante. Uma forma de reduzir o tempo gasto pelo SGA neste passo de ajuste envolveria a redução do número de combinações de parâmetros, aumentando o tamanho dos incrementos do ajuste. Porém, esta decisão poderia ajustar os parâmetros do SGA de maneira imprópria, resultando em baixo desempenho no problema a ser avaliado.

Tabela 7.2: Resultados do ajuste dos parâmetros.

Exp.	GEO (τ)	GEO$_{var}$ (τ)	SGA ($popsize, p_c, p_m$)
1	3	7,75	1000, 0,7, 0,01
2	0,75	1	100, 0,9, 0,019
3	0,75	1,75	1000, 0,8, 0,0175
4	0	0	10000, 0,9, 0,0145
5	0,75	0,75	10000, 0,7, 0,01
6	2,5	3,5	100, 0,8, 0,0175
7	3,25	7,25	10000, 0,9, 0,019

O processo de ajuste definiu a melhor configuração de parâmetros para cada algoritmo, e para cada SP. Após isto, as configurações mostradas na Tabela 7.2 foram aplicadas nos algoritmos e estes foram executados em cada SP 2000 vezes que, por sua vez, foram divididas em 20 blocos de 100 execuções para que os resultados fossem mostrados em uma escala de 0 a 100. Cada execução teve como limite o valor máximo de 100000 avaliações da *Similarity*, exceto os SPs 1 e 7, que tiveram seu limite ajustado em 400000 e 800000 avaliações, respectivamente. Esta decisão foi tomada para que o desempenho dos algoritmos durante a geração dos dados de teste fosse melhor analisado, visto que estes SPs são os mais complexos de todo o conjunto de SPs de acordo com a Tabela 7.1. Todos os gráficos na próxima seção mostram a média destes 20 blocos de 100 execuções e, além disso, para desenhar os gráficos a percentagem total de cobertura dos caminhos foi obtida a cada 100 avaliações da *Similarity*.

7.2.2 Análise dos resultados

O primeiro conjunto de gráficos na Figura 7.3 mostra a percentagem média de cobertura em função dos dados de teste gerados para quatro dos sete SPs da Tabela 7.1. Estes SPs são programas com uma complexidade ciclomática média de aproximadamente 5, além dos *loops* e vetores. Nos SPs 3 e 6, todos os algoritmos alcançaram rapidamente

a cobertura total. No entanto, um detalhe em *zoom* mostra que no SP 3 o GEO_{var} alcançou praticamente a cobertura total gerando muito menos dados de teste que os outros algoritmos. Por outro lado, o SGA alcançou a cobertura total com menos dados de teste que o GEO para o SP 6, apesar dos 93% de cobertura obtidos pelo GEO_{var} no momento em que o SGA atingiu a cobertura total. Esta diferença é muito pequena (somente 7%) quando comparada aos resultados do SP 3, onde a diferença entre o GEO_{var} e o SGA foi de quase 35%.

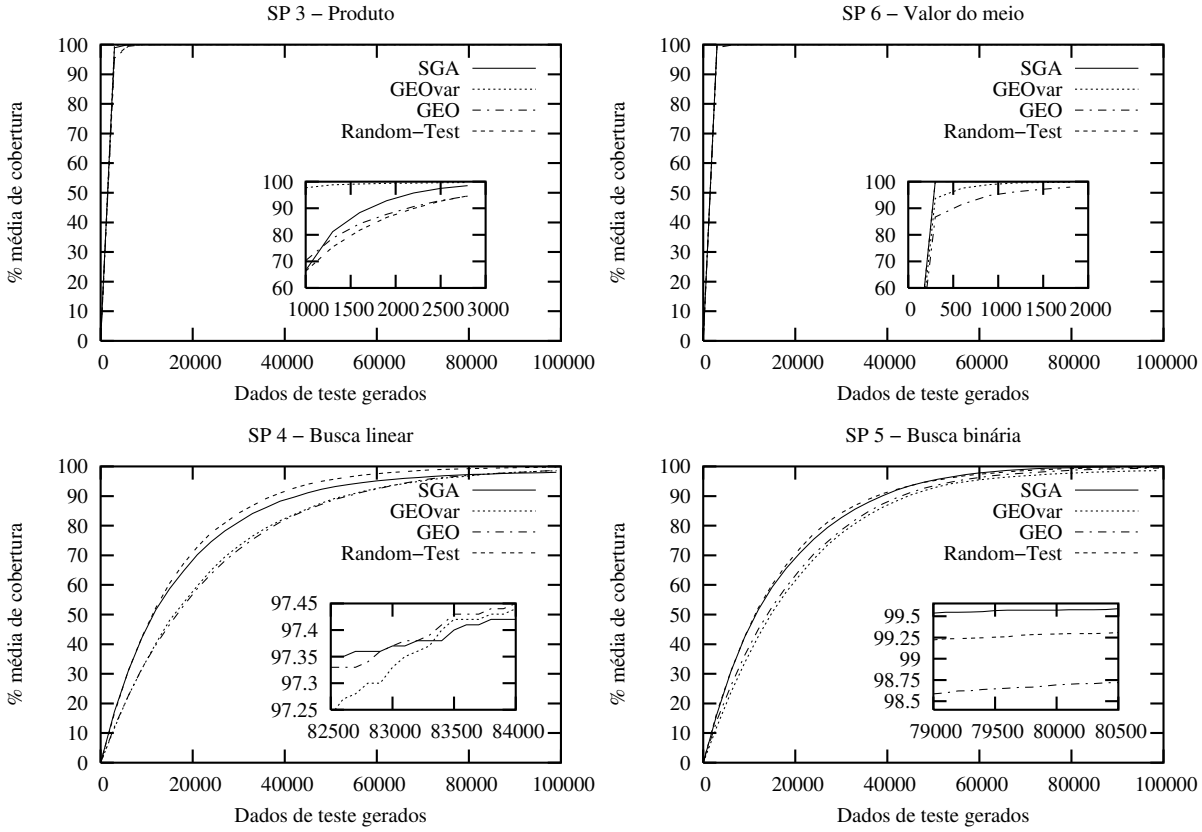

Figura 7.3: Evolução da % de cobertura média (1).

Os gráficos dos SPs 4 e 5 mostram um comportamento interessante. Embora a complexidade dos programas não seja muito alta, como mostrado anteriormente na Tabela 7.1, eles possuem construções envolvendo *loops* e vetores, o que pode dificultar a busca por dados de teste que satisfaçam determinadas condições em decisões ao longo do código. O perfil das curvas para os algoritmos mostra que a busca não convergiu rapidamente (para a cobertura total) mesmo utilizando variáveis de projeto com um domínio 4 vezes menor do que aquelas dos SPs 3 e 6. Este comportamento mostra que a geração de dados de teste para os SPs 4 e 5 foi mais difícil do que para os SPs 3 e 6. O desempenho do

Random-Test foi tão bom quanto o do SGA, inclusive melhor em algumas vezes. Este resultado confirma que as dificuldades relativamente baixas provenientes de SUTs simples podem fazer com que abordagens extremamente simples, como a geração aleatória de dados de teste, tenham um desempenho superior a outras abordagens [MMS01]; podendo inclusive sugerir que a abordagem é a melhor para problemas reais, o que não é verdade.

Para o SP 4, o gráfico deixa claro que o *Random-Test* teve um desempenho melhor que os outros algoritmos. Embora o SGA tenha adquirido uma vantagem de aproximadamente 10% de cobertura sobre o GEO na marca de 25000 dados de teste, o GEO ultrapassou-o na marca de 82900 dados de teste, e após 100000 dados de teste gerados a diferença entre o GEO e o SGA foi de 0,7%, e 0,56% para o GEO_{var}. No SP 5, a vantagem do SGA não ultrapassou 5% de cobertura, encerrando em 0,49% quando comparado ao GEO, e 1,3% ao GEO_{var}. Também é relevante explicar porque o desempenho do GEO e do GEO_{var} foi extremamente semelhante para estes SPs. Note que os SPs 4 e 5 são os programas de busca linear e binária, e que somente a chave de busca seria gerada pelos algoritmos. Neste caso há somente uma variável de projeto, e por esta razão o funcionamento do GEO_{var} é idêntico ao do GEO. Por outro lado, eles iniciaram a geração de dados de teste partindo de soluções diferentes, o que explica a diferença mínima entre seus resultados. Observe também que o tamanho da população do SGA para esses SPs foi 10000, o que poderia explicar o bom desempenho do SGA nelas. Por outro lado, mesmo iniciando a busca a partir de uma população com somente 14 indivíduos, o que é 714 vezes menor que o número de indivíduos da população do SGA, o GEO e o GEO_{var} foram melhores que o SGA para o SP 4 e praticamente tão bom quanto ele para o SP 5. Para o SP 2, os resultados da Figura 7.4 mostram um comportamento muito semelhante ao SP 3. Apesar disso, no SP 2 o SGA gerou menos dados de teste que o GEO para alcançar cobertura total. Além disso, a diferença entre eles chegou a quase 30% nos primeiros dados de teste gerados.

O segundo conjunto de gráficos, mostrado na Figura 7.5, avalia os algoritmos para os SPs 1 e 7, cujos programas possuem uma complexidade média 2,4 vezes maior do que os do primeiro conjunto. O SP 1 é o programa *simplified triangle*, e os resultados mostram vários pontos interessantes:

1. Todos os algoritmos cobriram 50% dos caminhos-alvo nos primeiros passos da geração dos dados de teste. Dois caminhos-alvo (triângulo escaleno e "não é um triângulo") são muito fáceis de serem cobertos, o que explica este fato;
2. O *Random-Test* alcançou rapidamente uma cobertura de 75% dos caminhos-alvo, o que explica seu desempenho alto nos primeiros 80000 dados de teste gerados;
3. O desempenho do *Random-Test* não ultrapassou a marca de 75% de cobertura pelo fato de ele não ter encontrado dados de teste para cobrir o caminho alvo do triângulo equilátero. Este resultado confirma que programas mais complexos, ou

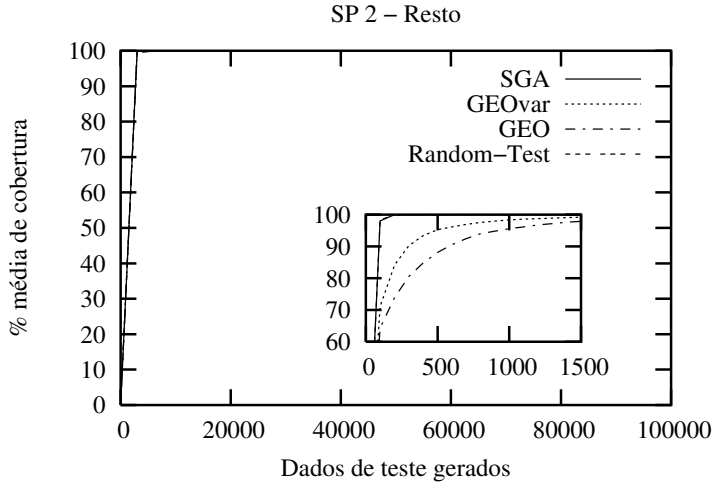

Figura 7.4: Evolução da % de cobertura média (2).

coberturas mais complexas, têm grandes chances de apresentar problemas para a abordagem de geração de dados de teste aleatória [MMS01]. Além disso, o valor alto para o parâmetro τ para este SP indica que este problema exige uma busca mais determinística pelas soluções, o que não pode ser feito, de fato, utilizando uma busca aleatória;

4. O GEO_{var} alcançou uma cobertura melhor no início do processo comparado ao GEO, mas o GEO ultrapassou-o na marca de 275000 dados de teste. Isto foi surpreendente, visto que o GEO_{var} sempre se manteve à frente do GEO durante a geração dos primeiros dados de teste;
5. Ao final do processo, com 400000 dados de teste gerados, a cobertura final do GEO foi praticamente a mesma do SGA. No entanto, a inclinação da curva do GEO mostrada no gráfico dá indícios de que ele o SGA nos próximos passos de geração dos dados de teste;
6. O bom desempenho inicial do SGA está provavelmente relacionado ao tamanho inicial de sua população, que é quase 21 vezes maior que a do GEO.

O SP 7 trouxe dificuldades para todos os algoritmos. Além de encontrar dados de teste para cobrir os mesmos tipos de triângulo do SP 1, este programa de estudo classifica um triângulo escaleno em três categorias de acordo com seus ângulos internos (reto, acutângulo e obtusângulo). Mesmo definindo um limite alto de 800000 para o número máximo de dados de teste gerados, nenhum algoritmo alcançou cobertura total, ou mesmo ultrapassou 70% de cobertura. Há algumas razões para esta cobertura final baixa. Primeiramente, os caminhos-alvo que não foram cobertos foram o do triângulo equilátero e do triângulo reto. A busca de dados para cobrir o caminho alvo do triângulo equilátero

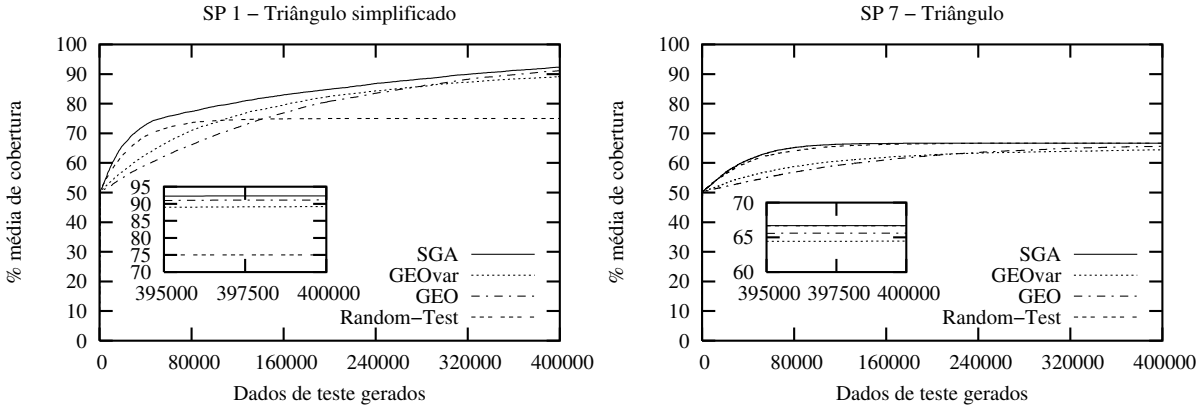

Figura 7.5: Evolução da % de cobertura média (3).

já havia apresentado dificuldades para os algoritmos no SP 1, mas a do triângulo reto foi mais difícil ainda. Este triângulo possui um ângulo de 90 graus, e seus lados a, b e c precisam satisfazer o Teorema de Pitágoras $a^2 + b^2 = c^2$, sendo c o maior lado. Este tipo de restrição é chamada de igualdade não-linear, um dos tipos mais difíceis em problemas de otimização.

Em segundo lugar, o conjunto de caminhos-alvo é implementado utilizando a estrutura de dados de fila, com política FIFO, com o caminho alvo correspondente ao triângulo reto precedendo o do triângulo equilátero. Sendo assim, o GEO e SGA estavam gerando dados de teste na tentativa de cobrir o caminho alvo do triângulo reto primeiro. Porém, além de ser muito difícil encontrar dados de teste para cobri-lo, os dados para este caminho em particular são muito diferentes daqueles para o triângulo equilátero, reduzindo de maneira significativa a possibilidade de ocorrer a cobertura colateral. Sendo assim, o esforço computacional na busca por dados de teste para cobrir um caminho, por exemplo, pode resultar na cobertura de mais de um caminho.

A Tabela 7.3 apresenta um resumo das percentagens de cobertura dos caminhos-alvo obtidas pelos algoritmos no final do processo de geração de dados. Os itens marcados em negrito indicam o algoritmo que obteve a maior percentagem de cobertura para cada SP.

Tabela 7.3: Percentagem de cobertura obtida pelos algoritmos em cada SP.

SP	SGA	GEO	GEO_{var}	*Random-Test*
1	**92.33%**	91.11%	89.16%	75%
2	100%	100%	100%	100%
3	100%	100%	100%	100%
4	98.07%	98.71%	98.58%	**99.78%**
5	**99.94%**	99.43%	98.62%	99.7%
6	100%	100%	100%	100%
7	**66.67%**	**66.67%**	65.58%	**66.67%**

O gráfico da Figura 7.6 apresenta os dados de tempo consumido pelos algoritmos nas 2000 execuções, para cada SP. A Tabela 7.4, além de apresentar os mesmos dados, marca em negrito o algoritmo com o menor consumo de tempo para cada SP. A princípio, vale a pena observar que um dos motivos para o grande tempo consumido pelo SP 7 (variando de 13,50h a 72,79h) foi o limite máximo de 800000 avaliações da *Similarity*; a dificuldade extrema para encontrar dados de teste para cobrir o caminho alvo relativo ao triângulo reto fez com que, quase sempre, as 800000 avaliações fossem realizadas. Note que tanto o GEO quanto o GEO_{var} consumiram muito menos tempo do que o SGA para este SP, sendo que a diferença final entre o GEO e o SGA foi de aproximadamente 28h, ou seja, mais de 1 dia. Este resultado dá fortes indícios de que o GEO é um algoritmo com um custo computacional bem menor que o SGA.

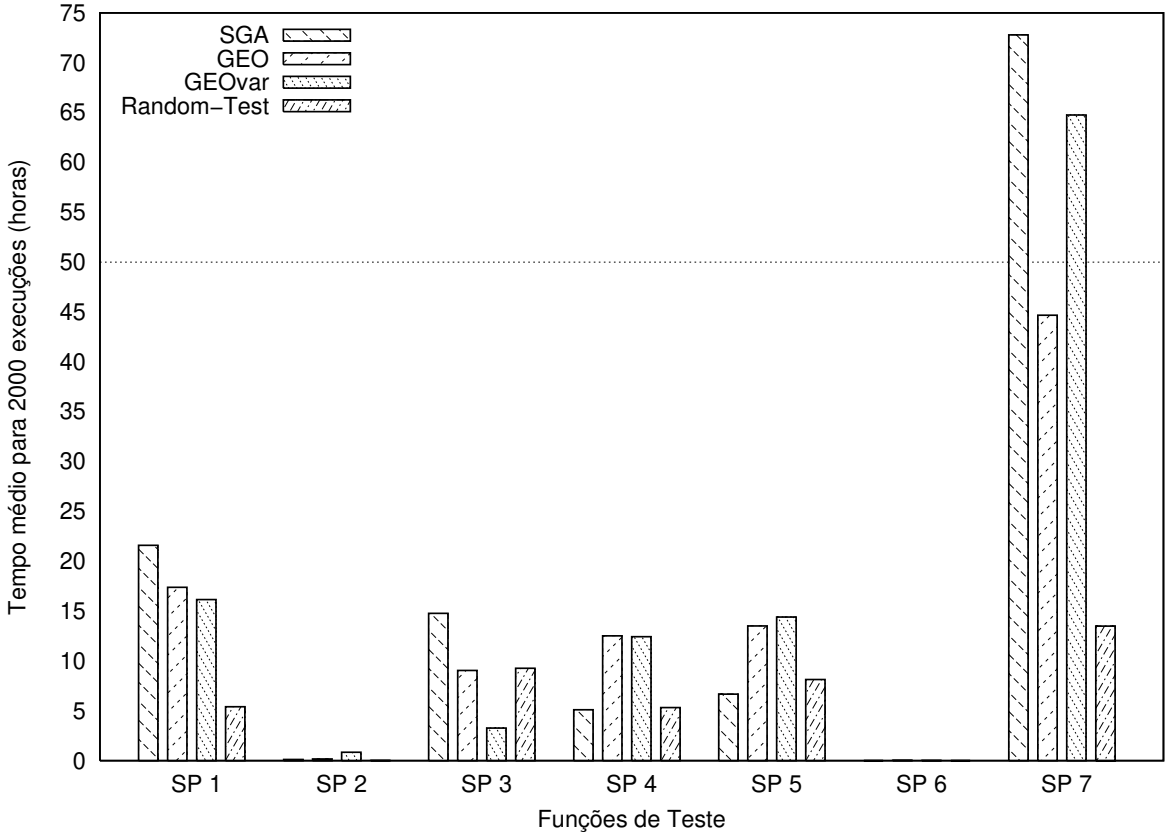

Figura 7.6: Tempo médio consumido pelos algoritmos em cada SP.

Um detalhe que contribui para este indício é que o SGA representa sua população em *popsize* vetores de n bits cada, enquanto que o GEO representa toda sua população em um único vetor com os mesmos n bits; ou seja, no caso do SP 7 onde a população do SGA é 10000, o SGA utilizou 9999 vetores a mais que o GEO, aumentando bastante o

Tabela 7.4: Tempo consumido pelos algoritmos em cada SP.

SP	SGA	GEO	GEO_{var}	*Random-Test*
1	21,59h	17,39h	16,15h	**5,41h**
2	0,12h	0,17h	0,85h	**0,02h**
3	14,78h	9,05h	**3,27h**	9,27h
4	**5,11h**	12,51h	12,44h	5,32h
5	**6,67h**	13,51h	14,40h	8,14h
6	0h	0,03h	0,02h	**0h**
7	72,79h	44,67h	64,75h	**13,50h**

consumo de memória, como mostra a Tabela 7.5[4]. Além disso, o SGA possui em seus três processos internos uma etapa de geração de um número aleatório seguido de um teste de uma condição, o que é uma operação cara, ainda mais quando se trata do processo de mutação, onde são feitas estas duas operações para cada bit de cada indivíduo. Por exemplo, sabendo que o SGA possui 10000 indivíduos, e que cada indivíduo representa 3 variáveis de 16 bits cada, estas operações serão feitas 480000 vezes, em somente uma iteração do algoritmo. Observe também que o *Random-Test* consumiu menos tempo que todos os algoritmos, mesmo porque sua lógica interna se resume somente a gerar números aleatórios em cada iteração, o que é algo bem simples.

Tabela 7.5: Memória física utilizada pelos algoritmos em cada SP.

SP	SGA	GEO	GEO_{var}	*Random-Test*
1	17128[a]	16168	14552	12372
2	16340	16380	16204	14468
3	17988	16256	16124	14404
4	17964	16204	16160	14380
5	17976	16220	16268	14424
6	13244	14036	13540	10932
7	22438	16184	16228	14404

[a]Valores em *kbytes*, obtidos através do comando UNIX **top**.

No caso do SP 1, cada algoritmo (exceto o *Random-Test*) alcançou praticamente a mesma cobertura final média. Note, porém, que o GEO consumiu entre 4 e 5 horas a menos que o SGA. De acordo com a Tabela 7.6, o número médio de execuções (ou dados de teste gerados) deste SP pelo SGA foi 254065, enquanto que o GEO gerou 13813 dados de teste a mais; este resultado reforça as observações feitas no parágrafo anterior. O *Random-Test*, apesar de ter consumido cerca de 10h a menos que os outros algoritmos, não alcançou a cobertura total em nenhuma das 2000 execuções. Isto mostra que embora o método apresente eficiência em termos de consumo de tempo, o uso de uma busca completamente aleatória deixa muito a desejar quando o SUT requer dados muito específicos e difíceis de serem encontrados (como, por exemplo, os dados para o triângulo equilátero).

[4]Vale lembrar que a simples medição de consumo de memória realizada não é suficiente para comprovar que o GEO possui um custo computacional menor que o SGA

Tabela 7.6: Número médio de execuções de cada SP pelos algoritmos.

SP	SGA	GEO	GEO_{var}	*Random-Test*
1	254065	267878	**239991**	400000
2	50	633	372	**49**
3	1783	2319	**720**	2348
4	44248	51250	50804	**36486**
5	**47919**	52437	60884	50700
6	13	521	215	**12**
7	800000	**799713**	799718	800000

Os SPs que apresentaram menos problemas para os algoritmos foram os SPs 2 e 6, tanto que a cobertura total para todas as 2000 execuções foi alcançada de forma extremamente rápida. Para estes SPs, o pior resultado de todos foi o do GEO_{var} no SP 2; neste caso, as 2000 execuções consumiram 51 minutos ou, colocado de outra forma, cada execução consumiu em média somente 1,53 segundos. No caso dos SPs 4 e 5, o GEO teve um desempenho pior que o SGA e *Random-Test*. Isto pode ser explicado em parte pelo tamanho elevado da população do SGA — 10000 indivíduos —, o que permitiu a avaliação direta, logo no início do processo, de 10000 candidatas a solução (possivelmente diferentes), sem que os processos internos do SGA fossem executados. Outro fator que contribuiu é a natureza aleatória deste problema, visto que os valores baixos para o parâmetro τ, como mostra a Tabela 7.2, indicam problemas que demandam uma busca próxima da aleatória, ou totalmente aleatória, ao invés de determinística.

Um resultado interessante foi o do SP 3. Neste SP o GEO_{var} teve um desempenho bem superior ao SGA, chegando a ser 4,5 vezes mais rápido que ele. Embora o SGA tenha executado o SUT menos vezes que o GEO, o tempo consumido por este (devido à complexidade de seus processos internos) foi vezes 1,6 vezes maior que o GEO. O número de execuções do SUT pelo GEO e *Random-Test* foi praticamente o mesmo, e o tempo consumido também. Note que a diferença de tempo entre eles foi de 0,22h, para uma diferença de 39 execuções. Este resultado indica que o GEO tem, em termos de consumo de tempo, um desempenho semelhante ao *Random-Test* quando o SUT é executado poucas vezes, mesmo tendo uma série de processos internos; nenhum tipo de experimento foi feito para avaliar a partir de quantas execuções este comportamento se altera, mas é certo de que ele se altera a partir de certo ponto em virtude dos resultados para o SP7. Para aquele SP, o número médio de execuções do SUT para todos os algoritmos foi praticamente o mesmo, porém o *Random-Test* consumiu de 31,17h a 59,29h a menos que os demais.

A partir dos resultados mostrados nas Tabelas 7.3, 7.4 e 7.6, é possível concluir que a avaliação do desempenho dos algoritmos nestes estudos de caso simples mostrou que a cobertura média alcançada pelo GEO é semelhante à do SGA em todos os SPs, e melhor do que a do *Random-Test* nos SPs mais complexos. A maior diferença em percentagem de cobertura foi 3,17% no SP 1, entre o GEO_{var} e o SGA. Embora o SGA tenha gerado os

dados de teste mais rápido nos SPs menos complexos (SP 2 ao SP 6), o GEO e o GEO_{var} consumiram muito menos tempo que ele para gerar os dados de teste para os SPs mais complexos (SP 1 e 7).

Capítulo 8

Estudo de Caso: Testando o OBDH-EXPEmulator

No intuito de avaliar a aplicabilidade do GEO para a geração automática de dados de teste para programas maiores e mais complexos, o mesmo foi aplicado para gerar dados para cobrir caminhos de uma aplicação real desenvolvida no Instituto Nacional de Pesquisas Espaciais (INPE), chamada de OBDH-EXPEmulator. A grande complexidade desta aplicação é importante de um ponto de vista prático, visto que é ela que validará a proposta desta dissertação de mestrado para programas reais que, em sua maioria, possuem módulos grandes e complexos.

Este capítulo inicia com uma descrição da estrutura do experimento realizado com o OBDH-EXPEmulator. A Seção 8.2 detalha os passos necessários para a preparação do experimento, o que também inclui o ajuste dos parâmetros dos algoritmos que serão utilizados para a geração de dados de teste, enquanto a Seção 8.3 trata do passo de execução do experimento. Por fim, os resultados obtidos são analisados na Seção 8.4, a última deste capítulo.

8.1 Descrição do experimento

O experimento irá testar o comportamento do OBDH-EXPEmulator ao receber comandos (mensagens) gerados automaticamente utilizando o GEO. Ele foi dividido em três etapas: preparação, execução e análise dos resultados. A preparação envolve o entendimento da aplicação a ser testada, a seleção e instrumentação das classes a serem testadas, bem como a seleção de caminhos a serem cobertos, além da definição das variáveis de entrada (domínio, tipo e precisão). Além disso, nesta etapa os algoritmos são integrados ao OBDH e os parâmetros dos mesmos são ajustados, quando aplicável. Estes passos são equivalentes às tarefas descritas na Seção 6.1 do Capítulo 6. A etapa de execução consiste

em configurar os algoritmos com a melhor combinação de parâmetros obtida com o seu ajuste na etapa anterior, e executá-los na tentativa de gerar dados para cobrir todos os caminhos selecionados.

Para a análise dos resultados, os critérios de avaliação utilizados foram os mesmos aplicados aos estudos de caso simples do capítulo anterior: cobertura média de caminhos, número de execuções do programa, e o tempo consumido durante a geração de dados de teste. Neste experimento, também foi considerado o tempo consumido no processo de ajuste de parâmetros dos algoritmos, quando aplicável. Dessa forma, será possível ter uma noção do tempo total consumido no processo de geração automática de dados de teste para cobrir caminhos de uma aplicação real relativamente complexa. Os algoritmos que foram comparados com o GEO foram: o SGA, o RT e uma implementação do algoritmo genético com *crossover* de dois pontos (AG-CD), cuja descrição encontra-se na Seção 3.3.4 do Capítulo 3. Todos os experimentos também foram executados em um Intel XEON CPU com 2.40GHz, quatro processadores e 1GB de RAM.

8.2 Preparação do experimento

8.2.1 A aplicação OBDH-EXPEmulator

O OBDH-EXPEmulator [dPE05] é uma aplicação que simula a comunicação entre um dispositivo chamado *On-Board Data Handling* (OBDH) e uma versão simplificada de um experimento científico específico, chamado de Observação de Raios Cósmicos Anômalos e Solares na Magnetosfera (ORCAS), lançado a bordo do satélite SACI-1 [INP]. A aplicação é composta de 20 classes Java, totalizando 2155 linhas de código, e possui métodos com complexidade ciclomática de McCabe [McC76] de até 27.

A Figura 8.1 ilustra a estrutura envolvendo o OBDH e a aplicação EXPEmulator, que implementa o experimento ORCAS. Observe que o canal de comunicação utilizado para o envio das mensagens, segundo a especificação da aplicação, é a porta serial RS-232 [Ele].

A Figura 8.2, por outro lado, ilustra o funcionamento básico da comunicação entre o OBDH e o experimento. Quando uma mensagem é produzida, esta é enviada ao EXPEmulator. O método **run()** é chamado e este, por sua vez, chama o método **waitMessage()**. Este último método pertence à classe *EOP.java*, e seu objetivo é ler a mensagem enviada pelo canal de comunicação byte a byte. Este método implementa a máquina de estados que representa o protocolo de comunicação OBDH-EXP. Caso a mensagem seja válida, o método **processMessage()** é chamado, direcionando a requisição do OBDH ao experimento (neste caso, o ORCAS). O método **processMessage()** pertence à classe *EXP.java*, que implementa os experimentos científicos. Ele envia uma requisição ao experimento que, por sua vez, retorna um resultado para o método **processMessage()**. Após o processa-

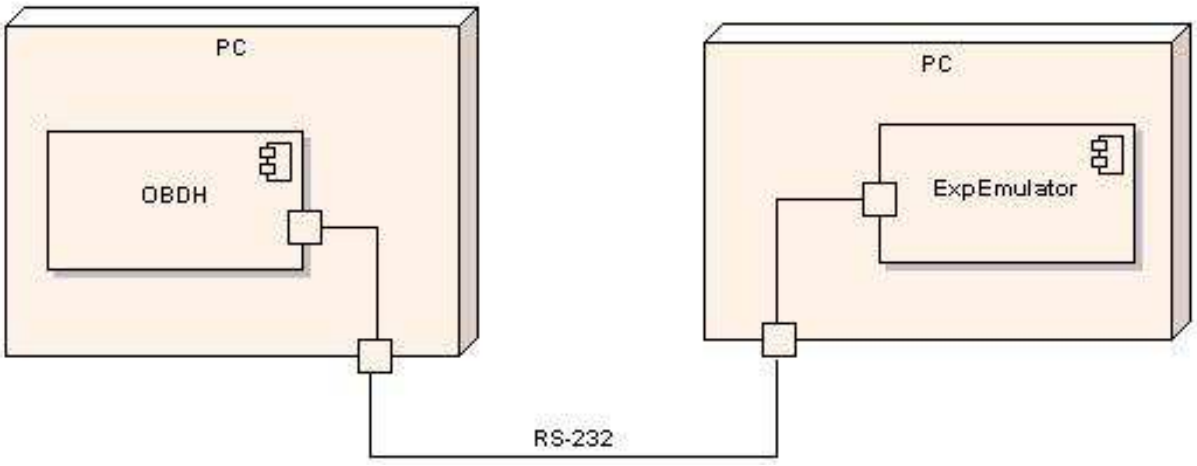

Figura 8.1: Canal de comunicação original entre o OBDH e o experimento.

mento da mensagem e a obtenção de uma resposta, esta pode ser enviada ao OBDH pelo experimento através do canal de comunicação, utilizando o método **sendMessage()** da classe *EOP.java*.

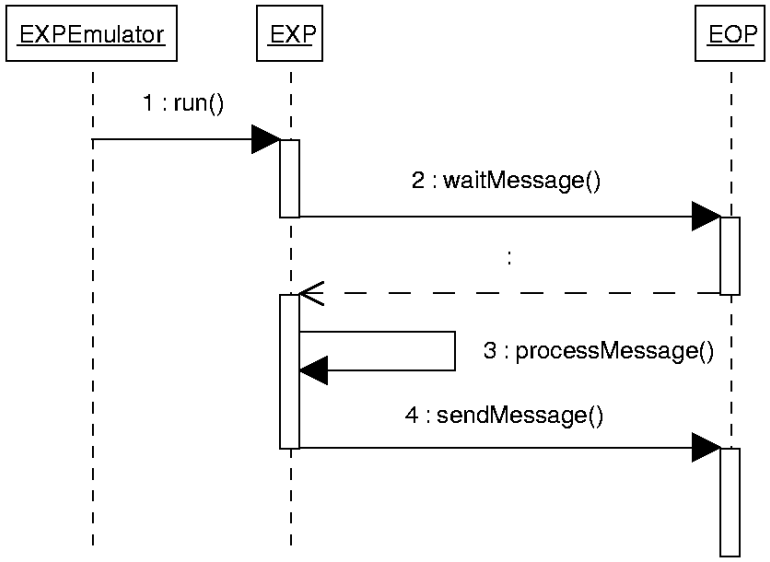

Figura 8.2: Processo de comunicação entre o OBDH e o experimento.

A mensagem enviada pelo OBDH ao experimento segue o formato mostrado na Figura 8.3a, enquanto a mensagem de resposta enviada do experimento para o OBDH segue o formato da Figura 8.3b. Ambas as mensagens possuem um total de 43 bytes, divididos entre quatro ou seis campos.

Detalhes a respeito das funcionalidades do protocolo encontram-se em [JMF98]. Estas

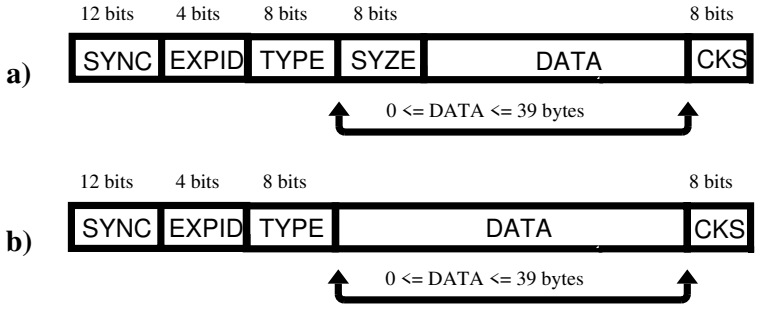

Figura 8.3: Formato das mensagens na comunicação entre OBDH e o experimento.

funcionalidades são requisitos necessários para manter em operação os experimentos a bordo de um satélite durante a fase de rotina da missão científica.

Quanto ao canal de comunicação, resultados preliminares indicaram que a porta serial RS-232 era um "gargalo", visto que muitas mensagens geradas e enviadas ao EXPEmulator não eram recebidas ou estavam incompletas. Uma solução para este problema foi estipular um tempo de espera entre o envio de cada mensagem; o tempo de espera adequado foi de 200ms, mas como o número de mensagens geradas e enviadas era demasiadamente grande, uma única simulação poderia levar mais de 100h para ser completada.

Uma alternativa para este problema foi realizar a comunicação utilizando a Invocação Remota de Métodos (em inglês, *Remote Method Invocation*, RMI) de Java. O RMI é uma das abordagens da tecnologia Java para prover as funcionalidades de uma plataforma de objetos distribuídos [WW06]. No entanto, o uso desta abordagem apresentou um desempenho pior que o obtido utilizando a porta serial RS-232.

Sendo assim, a comunicação entre o OBDH e o EXPEmulator foi feita através de uma chamada local a método, o que reduziu de maneira significativa o tempo consumido, o que será observado nas Seções 8.2.6 e 8.4. Note que esta decisão foi tomada visto que ela não interfere na avaliação do GEO como gerador de dados de testes. Vale ressaltar que a comunicação entre o OBDH e qualquer experimento utiliza um protocolo chamado OBDH-EXP [JMF98].

8.2.2 Variáveis de entrada

De acordo com a Figura 8.3, as mensagens enviadas ao experimento pelo OBDH possuem um total de 43 bytes, divididos em 6 campos: $SYNC$, $EXPID$, $TYPE$, $SIZE$, $DATA$ e CKS. Note que como o objetivo é cobrir caminhos da aplicação que simulem a recepção de mensagens, processamento e envio de respostas pelo experimento, o formato de mensagem a ser utilizado é o da Figura 8.3a. A descrição e os valores possíveis para estes campos encontram-se logo abaixo [JMF98]:

- O campo para sincronização $SYNC$ possui 12 bits e seu valor padrão é 0xEB9.
- O campo $EXPID$ identifica o experimento e possui 4 bits; no caso do ORCAS, o valor padrão é 0x2.
- O campo $TYPE$ possui 8 bits e identifica uma série de nove comandos que podem ser enviados ao experimento: restaurar o microcontrolador (0x01), enviar o *clock* (0x02), iniciar a aquisição de dados (0x03), interromper a aquisição de dados (0x04), transmitir dados (0x05), reconfigurar/alterar modo de operação (0x08), realizar *dump* de memória (0x1A), carregar dados na memória (0x1B) e carregar parâmetros (0x1F).
- O campo $SIZE$ é opcional e possui 8 bits; seu uso é obrigatório para os comandos de reconfigurar/alterar modo de operação, realizar *dump* de memória, carregar dados na memória e carregar parâmetros. Nestes casos, o campo $SYZE$ especifica o tamanho dos dados que estão sendo transmitidos no campo $DATA$, descrito logo a seguir.
- O campo $DATA$ também é opcional, sendo utilizado somente com os comandos especificados na descrição do campo $SIZE$, logo acima. O tamanho máximo para este campo é 304 bits, ou 38 bytes. No caso dos comandos de reconfigurar/alterar modo de operação e carregar parâmetros, o campo não ocupa mais do que 7 bits, uma vez que o maior valor para o campo $DATA$ é 0x48, ou seja, 72 em representação decimal.
- O último campo, CKS, tem como finalidade verificar a integridade da mensagem. Ele possui 8 bits e seu cálculo é feito utilizando uma equação descrita em [JMF98].

Baseando-se nestas informações, seriam necessárias seis variáveis de entrada (ou projeto), uma para cada campo da mensagem enviada pelo OBDH ao EXPEmulator. Porém, dois campos foram excluídos da geração automática: $SYNC$ e CKS. O primeiro campo foi excluído pelo fato dele ser um padrão para qualquer experimento. Já o campo de *checksum* foi excluído visto que ele é calculado automaticamente a partir dos demais campos da mensagem; dessa forma, o esforço de se gerar automaticamente um valor de CKS que combinasse com os demais campos da mensagem foi eliminado.

A eliminação destes dois campos fez com que sobrassem somente quatro variáveis de projeto: $EXPID$, $TYPE$, $SIZE$ e $DATA$. É importante ressaltar que o campo $TYPE$ determina o tamanho da mensagem, como mostra a Tabela 8.1.

Tabela 8.1: Relação entre o campo $TYPE$ e o tamanho da mensagem.

Domínio do campo	Tamanho da mensagem
[0,7]	4 bytes
[8,39]	6 bytes

O tipo destas variáveis deve ser inteiro, de acordo com a especificação do protocolo OBDH-EXP [JMF98]. As Tabelas 8.2 e 8.3 apresentam o nome das variáveis, seu tipo e domínio, para cada tipo de mensagem (4 bytes ou 6 bytes). É importante comentar que os valores dos domínios foram convertidos de valores em hexadecimal para valores inteiros.

Tabela 8.2: Variáveis de projeto para mensagens de 4 bytes.

Variável	Tipo	Domínio
$EXPID$	inteiro	[0, 3]
$TYPE$	inteiro	[0, 7]

Dado que somente os comandos de realizar *dump* de memória e carregar dados na memória utilizam mais do que 1 byte do campo $DATA$, o domínio deste campo foi reduzido de forma a atender prioritariamente aos comandos de reconfigurar/alterar modo de operação e carregar parâmetros. Note que no caso dos dois comandos relacionados a operações na memória, o campo $DATA$ estaria entre 36 e 38 bytes, muito distante de um somente 1 byte. Da mesma forma, o domínio do campo $SIZE$ foi alterado para atender aos comandos de reconfiguração/alteração e carregar parâmetros. A restrição imposta a estes campos permitiu representar as mensagens que utilizam os campos $DATA$ e $SIZE$ em 6 bytes (1 byte para cada campo), e as demais em 4 bytes, sendo 1 byte para cada campo também.

Observe que o domínio escolhido para cada variável de projeto cobre valores válidos e inválidos. No caso do domínio da variável $DATA$ na Tabela 8.3, os valores 0x00 e 0x01 estão relacionados ao comando de reconfiguração/alteração do modo de operação; estes dois valores foram representados dentro do domínio [45, 76], uma vez que nem todos os valores pertencentes a este domínio são utilizados pelo comando de carregar parâmetros. Para isso, foi utilizado um passo de conversão, associando o valor inteiro 55 ao valor 0x00, e 56 ao valor 0x01. Com isso, o domínio de entrada foi reduzido, o que aumenta a probabilidade de geração de dados com sucesso.

Tabela 8.3: Variáveis de projeto para mensagens de 6 bytes.

Variável	Tipo	Domínio
$EXPID$	inteiro	[0, 3]
$TYPE$	inteiro	[8, 39]
$SIZE$	inteiro	[0, 1]
$DATA$	inteiro	[45, 76]

Em um primeiro momento, todas as mensagens geradas pelo GEO eram de 6 bytes, utilizando os quatro campos. Porém um problema foi identificado utilizando esta aborgadem. O método **waitMessage()** do EXPEmulator implementa uma máquina de estados que, por sua vez, implementa o protocolo de comunicação entre o OBDH e qualquer outro

experimento. Cada transição na máquina de estados é efetuada após a leitura de 1 byte da mensagem enviada pelo OBDH. Independente do estado em que a máquina estiver, um byte inválido sempre ativa uma transição para o estado inicial da máquina (equivalente a uma reinicialização).

Para explicitar o problema em enviar mensagens utilizando o tamanho único de 6 bytes, suponha o seguinte cenário: o OBDH envia uma mensagem cujo campo $TYPE$ indica uma mensagem de 6 bytes. A máquina de estados lê o primeiro byte, correspondente ao campo $SYNC$, e efetua uma transição. O segundo byte (campo $EXPID$) é lido logo depois e uma nova transição é efetuada. A leitura do terceiro byte (campo $TYPE$) indica que a mensagem é de 6 bytes, e o próximo estado da máquina será um estado que espera valores de entrada válidos para o quarto byte, que se refere ao campo $SIZE$. A leitura do quinto byte (campo $DATA$) é feita, e o valor esperado para o sexto byte é um valor de *checksum* válido (campo CKS). Sabendo que o sexto byte é calculado automaticamente no envio da mensagem, no caso de mensagens cujo campo $TYPE$ indique o tamanho de 6 bytes, o valor de CKS será sempre válido.

Suponha agora que o OBDH envia uma nova mensagem com 6 bytes, mas com o campo $TYPE$ indicando uma mensagem cujo tamanho é 4 bytes. Partindo do estado inicial, a máquina de estados lê o primeiro e segundo byte normalmente. A leitura do terceiro byte (campo $TYPE$) identifica uma mensagem de tamanho de 4 bytes. Sabendo que 333 bytes já foram lidos previamente, o próximo byte a ser lido deve obrigatoriamente ser um campo CKS. No entanto, todas as mensagens enviadas possuem 6 bytes independente do valor do campo $TYPE$; neste caso, os valores gerados para o quarto byte, que deveria ser o CKS no caso de uma mensagem de 4 bytes, estarão sempre associados ao campo $SIZE$, como mostra a Figura 8.4. Como o domínio dos campos $SIZE$ e CKS são bem distintos, além do fato do CKS ser calculado automaticamente, as chances das mensagens de 4 bytes serem válidas são muito pequenas.

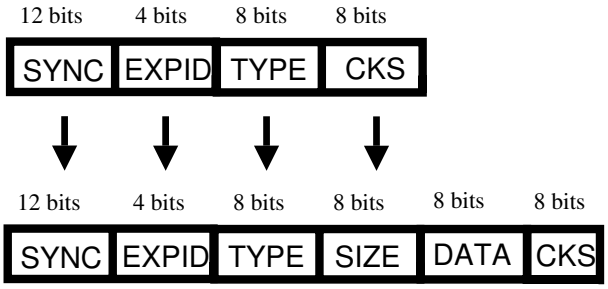

Figura 8.4: Correspondência entre o formato das mensagens de 4 bytes com as de 6 bytes.

Em virtude desta dificuldade, a solução adotada foi isolar as mensagens em duas categorias: as que utilizam os campos $SIZE$ e $DATA$ (mensagens de 6 bytes), e as que

não utilizam (mensagens de 4 bytes).

8.2.3 Seleção das classes para teste e instrumentação

Baseando-se na descrição do funcionamento da aplicação na Subseção 8.2.1, as classes selecionadas para instrumentação foram as que continham os métodos **waitMessage()**, **run()** e **processMessage()**: *EOP.java* e *EXP.java*, respectivamente. A classe *EOP.java* possui 316 linhas de código e seu método mais complexo é justamente o método **waitMessage()**, com complexidade ciclomática de 27[1]. Já a classe *EXP.java* possui 243 linhas de código e seu método com maior complexidade ciclomática é o **processMessage()**. O valor da complexidade, neste caso, é de 17. O método **run()**, desta mesma classe, possui complexidade ciclomática de 21. Estes métodos foram devidamente instrumentados utilizando um processo manual idêntico ao dos programas de estudo do Capítulo 7.

Observe que somente três métodos foram selecionados para teste, visto que eles são os responsáveis por simular o comportamento de um experimento desde o recebimento de uma mensagem do OBDH até o envio da resposta para o OBDH, indo de encontro aos objetivos deste experimento.

8.2.4 Seleção dos caminhos

A seleção dos caminhos a serem cobertos nos três métodos selecionados para teste foi feita a partir de um grafo de fluxo de controle interprocedimental (em inglês, *Interprocedural Control Flow Graph*, ICFG), uma proposta de Sinha et al [SHR01]. É importante lembrar que os caminhos não foram selecionados de acordo com um critério específico, visto que a seleção não é o foco desta pesquisa. Para cada método, um CFG foi construído de forma a representar os possíveis fluxos de execução no método. Observe, porém, que os métodos não serão testados isoladamente. Isto fica claro na Figura 8.5, que mostra o ICFG para o método **run()** da classe *EXP.java* e os nós e arestas marcadas com linhas tracejadas indicam chamadas a outros métodos. Em um ICFG, os CFGs são conectados através de arestas de chamada e entrada (*call, entry*), e saída e retorno (*exit, return*), mostrados em linhas tracejadas na Figura 8.5.

As Figuras 8.6 e 8.7 apresentam os CFGs para os métodos **waitMessage()** e **processMessage()**, respectivamente. Note que há uma quantidade enorme de caminhos possíveis ao analisar a combinação dos três métodos em questão.

A extração dos caminhos a partir do ICFG foi feita concatenando automaticamente caminhos extraídos manualmente dos CFGs dos métodos **waitMessage()** e **processMessage()**, além do ICFG para o método **run()**. Esta concatenação automática produziu

[1] A complexidade ciclomática foi medida utilizando a ferramenta Metrics [Met].

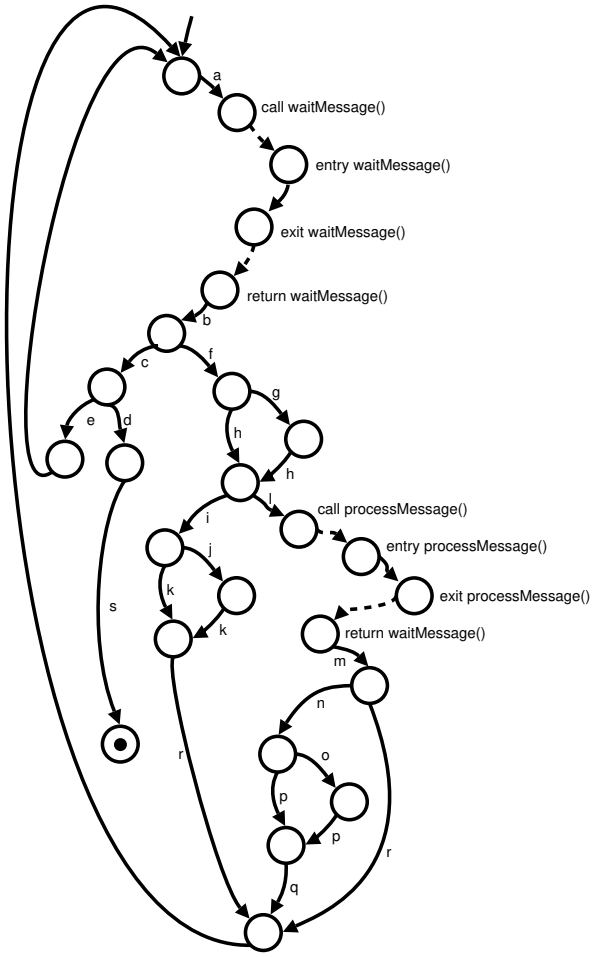

Figura 8.5: ICFG para o método **run()**.

1440 caminhos. Neste experimento, dentre todos estes caminhos foram selecionados somente 47, sendo 30 para mensagens de 4 bytes e 17 para mensagens de 6 bytes. Vale lembrar que no caso das mensagens de 6 bytes, não foram tratadas as mensagens relativas aos comandos de memória, o que explica o número baixo de caminhos para mensagens de 6 bytes. A redução de 1440 para somente 47 caminhos deve-se, principalmente, aos muitos caminhos não-executáveis que foram gerados automaticamente. A identificação deste tipo de caminho foi feita manualmente; além disso, execuções sucessivas do experimento também auxiliaram na identificação destes caminhos, uma vez que muitos deles nunca eram cobertos, independente do número de mensagens geradas e enviadas ao EXPEmulator.

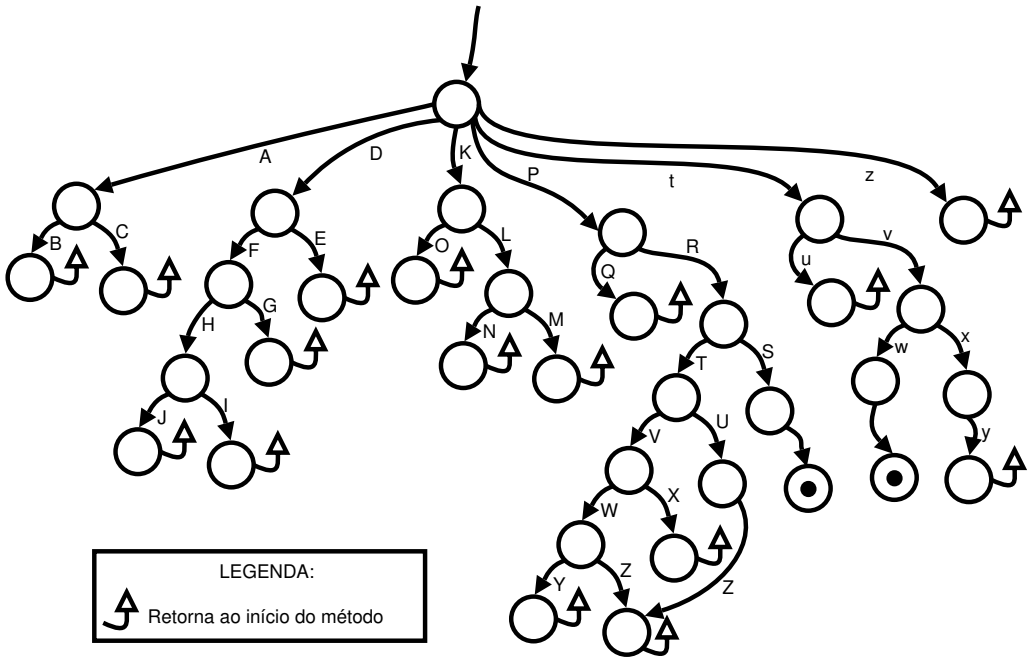

Figura 8.6: CFG para o método **waitMessage()**.

8.2.5 Integrando os algoritmos ao OBDH-EXPEmulator

Como foi visto na Subseção 8.2.1, o OBDH atua enviando mensagens ao EXPEmulator. Todo o comportamento que será testado pertence única e exclusivamente ao EXPEmulator, uma vez que o objetivo de testes neste experimento é cobrir um subconjunto de caminhos que envolvem a recepção, processamento e envio de resposta do experimento ao OBDH.

Sendo assim, os dados de teste que serão gerados deverão ser encapsulados em uma mensagem que, por sua vez, será enviada ao EXPEmulator. O formato da mensagem é o mesmo da Figura 8.3a. Resumidamente, o GEO e outras ferramentas para a geração de dados de teste realizarão a função de um *driver* de testes[2], enviando mensagens para o EXPEmulator com o objetivo de cobrir um subconjunto de caminhos pré-selecionados.

8.2.6 Ajustando os parâmetros dos algoritmos

Tal como nos experimentos com os estudos de caso simples do capítulo anterior, os parâmetros dos algoritmos foram ajustados de forma a maximizar o seu desempenho para a geração de dados de teste para o EXPEmulator. O único algoritmo que não exigiu ajuste foi o RT, uma vez que este não possui parâmetros ajustáveis. Tanto o GEO quanto

[2]Um *driver* de testes é um programa ou classe que aplica os casos de teste ao software em teste.

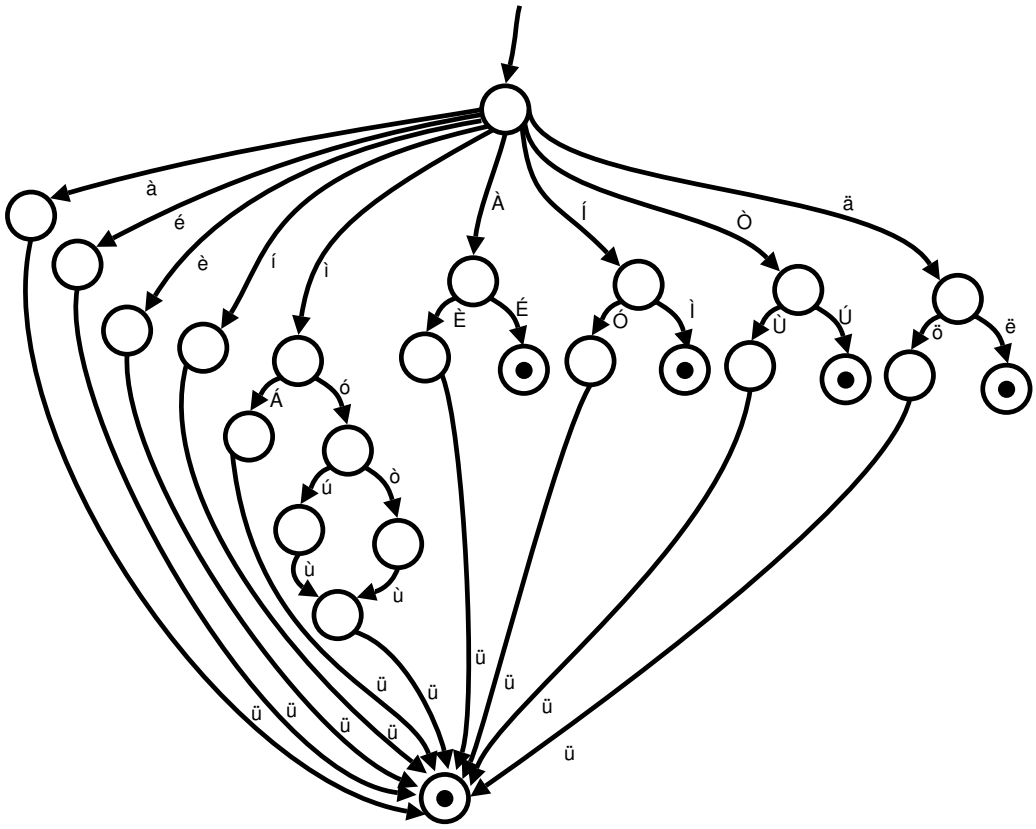

Figura 8.7: CFG para o método **processMessage()**.

os dois algoritmos genéticos — o SGA e o AG-CD — tiveram seus parâmetros ajustados. É importante comentar que o AG-CD possui os mesmos parâmetros ajustáveis do SGA: probabilidades de *crossover* (p_c) e mutação (p_m), além do tamanho da população (*popsize*), conforme visto na Seção 3.3.4 do Capítulo 3.

Para realizar o ajuste dos parâmetros, uma série de condições foram definidas:

- Cada algoritmo foi aplicado 10 vezes para cada combinação de parâmetros possível.
- O número máximo de avaliações da função objetivo *Similarity* foi limitado em 100000.
- Tanto o GEO quanto sua variação, GEO_{var}, iniciaram a busca a partir da mesma solução inicial, gerada aleatoriamente.
- O parâmetro τ (GEO e GEO_{var}) variou de 0 a 10, com incrementos de 0,5.
- Para o SGA e AG-CD:

$$\begin{cases} 100 \leq popsize \leq 10000 & \text{começando em 100 e com incrementos de } popsize * 10, \\ 0,6 \leq p_c \leq 1,0 & \text{começando em 0,6 e com incrementos de 0,1,} \\ 0,01 \leq p_m \leq 0,02 & \text{começando em 0,01 e com incrementos de 0,0025} \end{cases}$$

Todas estas condições foram aplicadas no ajuste dos algoritmos tanto para as mensagens de 4 bytes quanto para as de 6 bytes. Os resultados do processo de ajuste, incluindo o tempo consumido durante a execução de cada algoritmo, encontram-se na Tabela 8.4. Observe que, por mera coincidência, os melhores parâmetros para as mensagens de 4 bytes foram idênticos aos das mensagens de 6 bytes.

Tabela 8.4: Resultados do ajuste dos parâmetros.

	GEO (τ)	GEO$_{var}$ (τ)	SGA ($popsize, p_c, p_m$)	AG-CD ($popsize, p_c, p_m$)	RT
Parâmetros	0	0,5	100, 0,6, 0,01	100, 0,6, 0,01	–
Tempo[a] (em horas)	0,24	0,29	1,60	1,56	0
Tempo[b] (em horas)	0,26	0,31	1,80	1,83	0

[a]Mensagens de 4 bytes.
[b]Mensagens de 6 bytes.

Os resultados do ajuste mostram que o problema foi resolvido mais facilmente utilizando uma abordagem aleatória para a geração dos dados, o que é indicado pelos valores de τ para o GEO. Além disso, o tempo consumido para o ajuste do GEO foi bem menor do que para os algoritmos genéticos (em média, seis vezes menor). Uma razão para isto é o número de combinações de parâmetros possíveis para os algoritmos genéticos: estes possuem 75 ($3 * 5 * 5$) combinações, contra somente 21 do GEO. Como foi dito no Capítulo 5, no final da Seção 5.3, o único parâmetro ajustável do GEO faz com que ele tenha uma vantagem *a priori* frente aos algoritmos genéticos.

8.3 Execução do experimento

Na execução do experimento, os melhores parâmetros, mostrados na Tabela 8.4, foram atribuídos a cada algoritmo. Os algoritmos foram executados 10 vezes para cada tipo de mensagem (4 bytes e 6 bytes), sendo que o limite de avaliações da função objetivo *Similarity* foi de 2 milhões de avaliações. Este limite de execuções foi definido na tentativa de garantir a maior cobertura de caminhos possível. Ao longo de cada simulação, os dados do experimento foram coletados a cada 100 avaliações da *Similarity*. Os critérios de parada utilizados foram o número máximo de avaliações da função objetivo, ou a cobertura de todos os caminhos selecionados como alvo.

8.4 Análise dos resultados

Os primeiros critérios avaliados foram os de cobertura média de caminhos e número de execuções do programa (equivalente ao número de dados de teste gerados). No caso das mensagens de 4 bytes, a cobertura foi muito baixa, como mostra a Tabela 8.5 e

a Figura 8.8. A cobertura final para todos os algoritmos foi de 27,74%, e a evolução da cobertura de caminhos foi dividida em quatro momentos distintos. Cada momento representa um ponto que merece atenção, uma vez que permite diferenciar a cobertura alcançada pelos algoritmos ao longo do tempo com mais detalhes. Os itens em destaque na Tabela 8.5 representam os melhores valores dentre todos os algoritmos para o critério em análise.

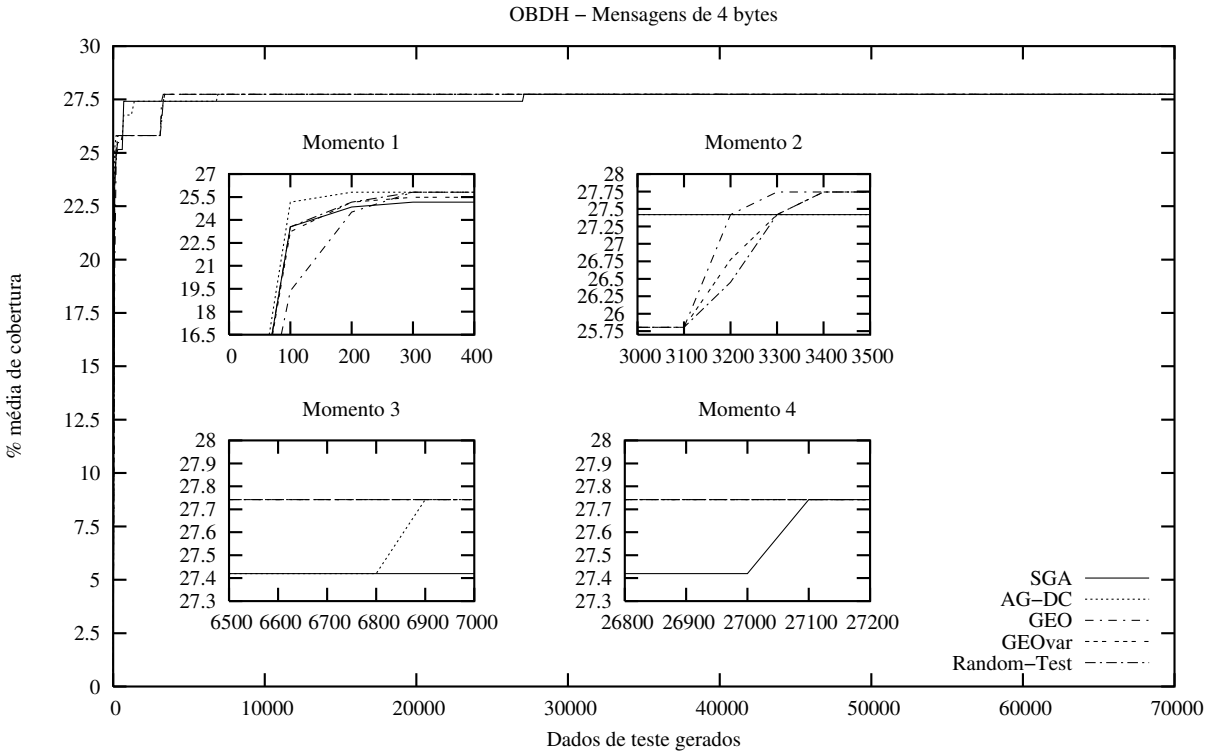

Figura 8.8: Comportamento para tamanho de mensagens de 4 bytes.

Tabela 8.5: Cobertura média dos caminhos-alvo (em cima) e número médio de execuções para cada tamanho de mensagem (embaixo).

Tamanho da mensagem	SGA	AG-CD	GEO	GEO_{var}	RT
4 bytes	27,74%	27,74%	27,74%	27,74%	27,74%
6 bytes	31,18%	31,18%	31,18%	31,18%	31,18%

Tamanho da mensagem	SGA	AG-CD	GEO	GEO_{var}	RT
4 bytes[a]	27100	6900	**3300**	3400	3400
6 bytes[a]	8700	**5700**	379500	222200	387700

[a]Número de execuções até alcançar a maior cobertura.

No "Momento 1", mostrado na Figura 8.8, fica claro que o algoritmo genético com *crossover* de dois pontos (AG-CD) obteve a maior cobertura nos 100 primeiros dados

de teste gerados, enquanto o GEO obteve a menor. No entanto, o "Momento 2"mostra que o GEO ultrapassou tanto o AG-CD quanto os demais algoritmos na marca de 3200 dados de teste. A cobertura máxima alcançada pelos algoritmos após a geração de 2 milhões de dados de teste[3] foi de 27,74%, e o GEO foi o primeiro algoritmo a alcançar esta marca com 3300 dados de teste gerados. A mesma marca foi atingida pelo GEO_{var} e Random-Test (RT) após a geração de 3400 dados de teste. O AG-CD somente alcançou a maior cobertura após a geração de 6900 dados de teste, como mostra o "Momento 3", ou seja, o AG-CD executou o EXPEmulator 3500 vezes a mais que o GEO. Para este tamanho de mensagem, o pior resultado foi do SGA. De acordo com o "Momento 4", este algoritmo executou o EXPEmulator 27100 vezes até alcançar a mesma cobertura dos demais algoritmos, o que é aproximadamente 8 vezes mais que o GEO e 4 vezes mais que o AG-CD.

A cobertura idêntica alcançada por todos os algoritmos ao final da geração de 2 milhões de dados de teste, como mostra a Tabela 8.5, indica que boa parte dos caminhos selecionados como alvo são não-executáveis. De fato, dentre os 17 caminhos que não foram cobertos para as mensagens de 4 bytes, um deles é não-executável e o restante são difíceis de serem cobertos, visto que dependem de uma fila de mensagens cujo estado interno não é conhecido pelos algoritmos, uma dificuldade apresentada por McMinn [MH03]. Outro ponto importante é que o resultado de 27,74% apresentado na Tabela 8.5 é a média de de 10 execuções, e que a melhor cobertura alcançada pelos algoritmos nas mensagens de 4 bytes foi de 45,16%, o que equivale à cobertura de 13 caminhos-alvo, e de 47,06% nas mensagens de 6 bytes, o que equivale à cobertura de 8. No caso das mensagens de 6 bytes, uma análise dos caminhos que não foram cobertos identificou 2 não-executáveis e outros 7 difíceis de se cobrir, pela mesma razão apresentada no caso das mensagens de 4 bytes.

A cobertura baixa também pode ser atribuída, em parte, às características da função objetivo utilizada; por exemplo, a *Similarity* não consegue diferenciar dois dados de teste que cobrem o mesmo caminho, o que pode fazer diferença no processo de geração e evolução dos dados. Watkins e Hufnagel [WH06] apresentam outras considerações a respeito desta função objetivo.

A Figura 8.9 apresenta os resultados de cobertura para as mensagens de 6 bytes. Note que a percentagem de cobertura para este tamanho de mensagem foi maior, mas vale lembrar que o número de caminhos selecionados como alvo neste caso foi somente 17, como visto na Seção 8.2.4, contra os 30 caminhos selecionados para as mensagens de 4 bytes. O valor máximo de cobertura média atingido por todos os algoritmos foi 31,18%. O "Momento 1"mostra que todos os algoritmos alcançaram praticamente a mesma cobertura nos primeiros 100 dados de teste gerados; porém, após isto, o AG-CD se destacou

[3]O gráfico mostra somente até a marca de 70000 dados de teste, uma vez que a cobertura total não sofreu alterações após a marca de 27200 dados de teste.

alcançando a cobertura de 31,18% com somente 5700 dados de teste. No caso dos demais algoritmos (exceto o SGA, cujo resultado foi semelhante ao AG-CD), o "Momento 2"mostra o momento em que o GEO ultrapassa o RT, na marca de 2600 dados de teste; este momento também deixa claro que o GEO_{var} teve o melhor resultado dentre o GEO e RT.

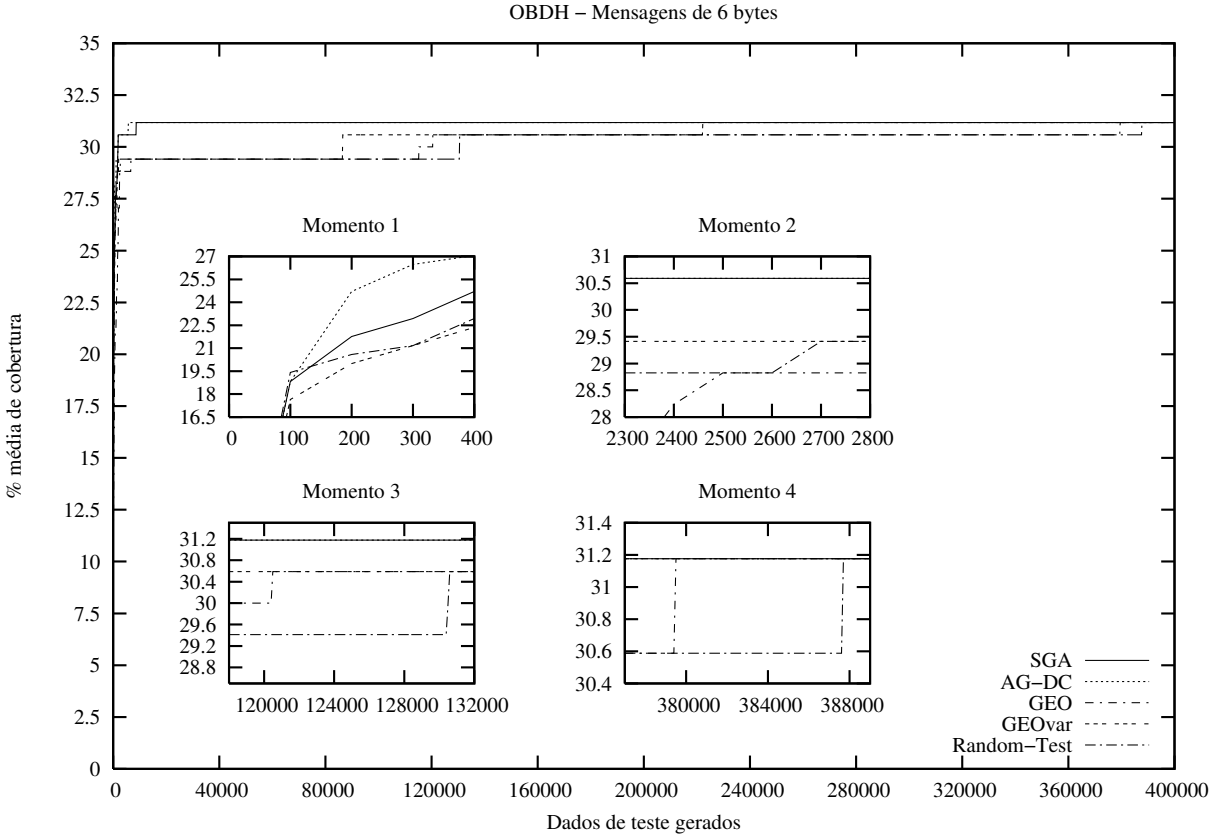

Figura 8.9: Comportamento para tamanho de mensagens de 6 bytes.

O "Momento 3"mostra que o GEO alcançou o GEO_{var} antes do RT, gerando cerca de 10000 dados de teste a menos. Já o "Momento 4"mostra o ponto em que o GEO e o RT alcançaram a melhor cobertura. Observe que para este tamanho de mensagem, o número de dados de teste gerados foi expressivamente maior do que o número para as mensagens de 4 bytes (no caso do GEO, GEO_{var} e RT). Isto é algo esperado, uma vez que este tamanho de mensagem utiliza duas variáveis de projeto a mais, aumentando o número de combinações possíveis dos dados de entrada, conseqüentemente aumentando a dificuldade de gerar dados que satisfaçam todas as restrições de um caminho alvo qualquer.

A análise dos algoritmos segundo o critério de tempo consumido foi feita tendo como base os resultados da Tabela 8.6 e da Figura 8.10. Vale lembrar que, na tabela, os itens

destacados indicam os melhores resultados dentre todos os algoritmos para cada tamanho de mensagem.

Tabela 8.6: Tempo consumido em 10 simulações para cada tamanho de mensagem.

Tamanho da mensagem	SGA	AG-CD	GEO	GEO_{var}	RT
4 bytes	8,76h	8,69h	3,72h	3,76h	**1,46h**
6 bytes	9,67h	10,20h	3,71h	3,78h	**1,22h**

Várias observações interessantes podem ser feitas a partir desta tabela e figura:

1. Tanto o SGA quanto o AG-CD consumiram mais tempo que o GEO. No caso das mensagens de 4 bytes, eles foram aproximadamente 2,3 vezes mais lentos que o GEO, e aproximadamente 2,7 vezes (na média) mais lentos para as mensagens de 6 bytes. Isto é explicado por toda a lógica interna do SGA e do AG-CD. Por exemplo, a operação de mutação nestes dois algoritmos é feita para cada bit do indivíduo; logo, em 2 milhões de execuções, para um indivíduo representado por 20 bits e membro de uma população de 100 indivíduos, o teste da mutação (ver Seção 3.3.4 do Capítulo 3) seria feito 4 bilhões de vezes.

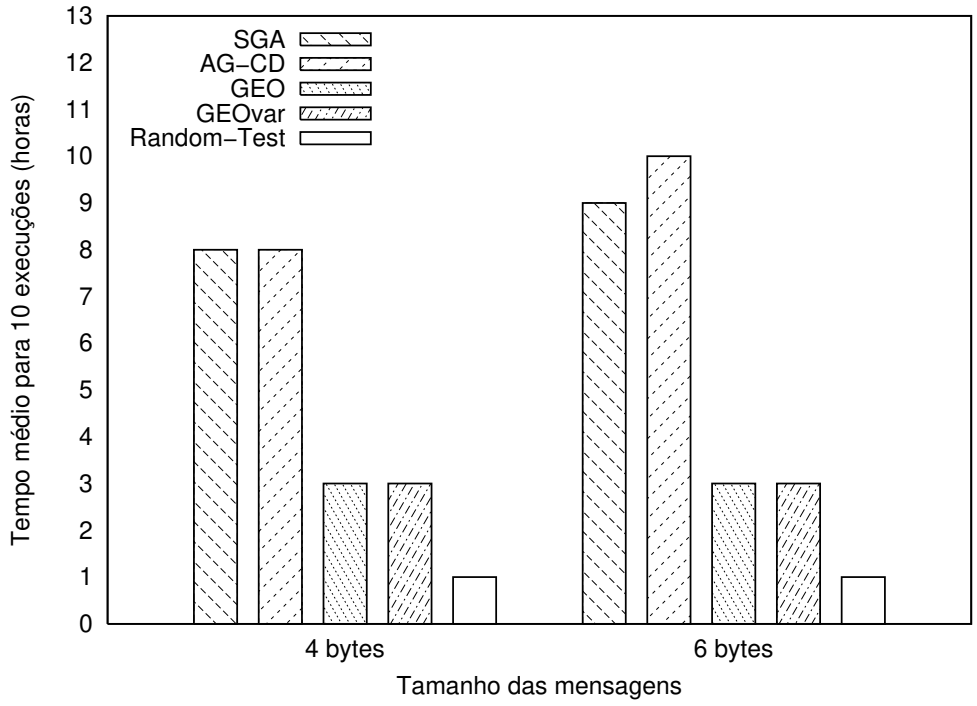

Figura 8.10: Tempo consumido em 10 simulações.

2. O tempo consumido pelo GEO e GEO_{var} foi praticamente o mesmo para ambos os tamanhos de mensagem. Uma justificativa para isto é que, mesmo o GEO_{var} trabalhando com cada variável de projeto "isoladamente"(ver Seção 5.3 do Capítulo 5),

os valores muito baixos do parâmetro τ fizeram com que a operação interna de escolha dos bits para sofrer mutação fosse feita rapidamente. Esta operação interna é repetida até que tenham sido escolhidos os bits para sofrer mutação (um para cada variável), e quanto maior o valor de τ, mais difícil é a escolha do bit, uma vez que somente os melhores bits serão confirmados para sofrer mutação. No caso deste experimento, os valores baixos de τ (0 e 0,5) fizeram com que praticamente qualquer bit escolhido fosse confirmado para sofrer a mutação, reduzindo o *overhead* deste passo do GEO.

3. O tempo consumido pelo RT foi o menor dentre todos os algoritmos, e a cobertura final alcançada por ele foi tão boa quanto a dos outros algoritmos analisados. Como foi dito no item anterior, os valores de τ para o GEO já indicavam que o problema em estudo seria resolvido utilizando uma abordagem aleatória, explicando em parte o sucesso do RT. O seu sucesso também é atribuído ao domínio pequeno das variáveis de projeto, tanto para as mensagens de 4 bytes quanto para as de 6 bytes. De acordo com Michael et al [MMS01], domínios pequenos reduzem o espaço de busca e aumentam as chances de sucesso de uma abordagem completamente aleatória.

Tabela 8.7: Tempo total consumido pelos algoritmos no experimento.

Atividades	SGA	AG-CD	GEO	GEO_{var}	RT
Ajuste de parâmetros[a]	3,4h	3,39h	0,5h	0,6h	-
Simulações[a]	18,43h	18,89h	7,43h	7,54h	2,68h
Total	21,83h	22,28h	7,93h	8,14h	2,68h

[a]Tempo para as mensagens de 4 e 6 bytes.

Concluindo, a Tabela 8.7 apresenta o tempo total consumido no experimento, para cada algoritmo. Como era de se esperar, o RT consumiu o menor tempo, mesmo porque este não necessita de ajuste de parâmetros, sua lógica interna é extremamente simples, o domínio das variáveis de projeto era pequeno e os valores de τ indicaram que o problema de testes seria resolvido utilizando uma abordagem aleatória. Dentre os algoritmos evolutivos, tanto o GEO quanto o GEO_{var} consumiram, de forma significativa, menos tempo que os algoritmos genéticos utilizados. Mesmo no caso das mensagens de 6 bytes, onde o GEO alcançou a mesma cobertura que o SGA e o AG-CD após a geração de mais de 200000 dados de teste (um número consideravelmente maior do que o do SGA e AG-CD), o tempo consumido foi menor.

Capítulo 9

Conclusões e trabalhos futuros

Esta dissertação avaliou, pela primeira vez, o desempenho de um novo algoritmo evolutivo, a Otimização Extrema Generalizada (em inglês, *Generalized Extremal Optimization*, GEO), para a geração dinâmica e automática de dados de teste para programas com ou sem *loops*. O tipo de teste aplicado foi o orientado a caminhos, onde um subconjunto de caminhos do SUT foi selecionado aleatoriamente para ser coberto. A evolução dos dados de teste para cobrir um caminho foi conduzida por uma função objetivo chamada *Similarity*, avaliando a distância do caminho coberto para o caminho alvo.

O desempenho do GEO foi avaliado através de estudos de caso, representado por SUTs simples e bem conhecidos na literatura e por uma aplicação real desenvolvida pelo Instituto Nacional de Pesquisas Espaciais (INPE). A análise envolveu também a comparação do GEO com outros três algoritmos: a geração aleatória de dados de testes, o algoritmo genético simples e uma variação dele, o algoritmo genético com *crossover* de dois pontos. Os critérios para a avaliação dos algoritmos foram a cobertura média de caminhos, o número de execuções do SUT (equivalente ao número de dados gerados), e o tempo consumido durante a geração dos dados de teste.

Os resultados mostraram que o GEO é um algoritmo evolutivo competitivo com os dois algoritmos genéticos analisados, para a geração de dados de teste para cobrir caminhos em programas. Para os critérios de avaliação de cobertura média de caminhos e número de execuções do SUT, o GEO foi melhor em alguns casos e ligeiramente pior em outros quando comparado aos algoritmos genéticos. No entanto, ele consumiu menos tempo na geração dos dados de teste quando o SUT era mais complexo. Além disso, o tempo para o ajuste de parâmetros para o GEO foi significativamente menor que o consumido pelo SGA e AG-CD. Estes resultados com relação ao critério de avaliação do tempo consumido são importantes, uma vez que é esperado que os problemas tratados em aplicações reais sejam mais complexos. Além da vantagem com relação ao tempo, o ajuste do parâmetro τ do GEO fornece uma informação importante com relação ao problema: o grau de

determinismo a ser aplicado na busca. Dessa forma, se o valor de τ for muito próximo de zero, uma sugestão é utilizar o *Random-Test* pois praticamente não há determinismo na busca pelos dados de teste.

Em suma, o GEO nem sempre oferece a melhor cobertura de caminhos com o menor número de execuções do SUT, mas o faz consumindo significativamente menos tempo se comparado ao SGA e AG-CD nos problemas mais complexos, cujas variáveis de projeto possuem um domínio grande. É importante comentar que, como a maioria dos algoritmos genéticos utilizados até o momento para gerar dados de teste são muito semelhantes ao SGA, pode ser dito que o GEO, ou outras variações dele, também é potencialmente competitivo, de uma maneira geral, com estes tipos de algoritmos.

O bom desempenho do *Random-Test* na aplicação real poderia indicar que a abordagem aleatória é a mais recomendada em todos os casos, por ser extremamente simples e rápida. Porém, as características da aplicação real utilizada no estudo de caso (domínio das variáveis de projeto pequeno e grau de determinismo na busca por dados inexistente) favoreceram o uso do *Random-Test*. Note que nem sempre ele foi o melhor algoritmo, principalmente nos problemas com complexidade ciclomática elevada e com domínio grande. Nesta aplicação em específico, o problema é que o valor dos dados de teste variam conforme o estado do sistema, que não é visível (a função objetivo utilizada não capturou esta informação) e, conseqüentemente, não é levado em conta na geração dos dados pelo GEO e os outros algoritmos.

A função objetivo utilizada nos estudos de caso chama a atenção pelo fato de trabalhar com o código do SUT em um nível de abstração mais alto, ocultando alguns detalhes de sua lógica interna. O que pode ser uma vantagem a primeira vista torna-se uma desvantagem durante a geração dos dados de teste, porque a abordagem utilizada pela *Similarity* não diferencia dados de teste que resultem na cobertura de um mesmo caminho. Dessa forma, tanto o GEO quanto os algoritmos genéticos trabalham evoluindo os dados de teste de maneira "cega", sem nenhuma informação que indique a melhor direção para a busca de dados, o que reforça a importância da escolha de uma boa função objetivo, adequada ao critério de teste utilizado.

9.1 Contribuições

A contribuição principal deste trabalho é o uso, pela primeira vez, de um algoritmo evolutivo proposto recentemente, o GEO, em um problema da Engenharia de Software, especificamente na área de Testes de Software. O desenvolvimento de heurísticas para a geração dinâmica de dados de teste é importante, e este trabalho mostra que o GEO é competitivo com outros algoritmos já utilizados e bem conhecidos na literatura, tornando-se uma opção muito interessante como ferramenta para geração automática de dados de

teste.

A avaliação do GEO quando aplicado a problemas de uma área diferente da de otimização pode ser considerada uma outra contribuição importante deste trabalho, mesmo porque boa parte da literatura na área de geração automática de dados de teste utiliza o algoritmo genético (AG) e suas variações. Mesmo sabendo que o AG tem processos internos que podem ser implementados utilizando diversas estratégias, o núcleo e a abordagem central dele são as mesmas. Como foi visto no Capítulo 5, a abordagem do GEO é diferente do AG, e ele ainda possui a vantagem de ser facilmente ajustável e consumir significativamente menos tempo na geração de dados de teste para problemas mais complexos.

Este trabalho também traz contribuições ao projeto CompGov[1], que tem a função de criar um repositório de componentes que possa ser utilizado por diversas entidades interessadas em empregar componentes com funcionalidades específicas prontas no desenvolvimento de seus sistemas de informação. A contribuição principal para este projeto é um protótipo de ferramenta para geração automática de dados de teste utilizando o GEO, o qual será evoluído para uma ferramenta que será integrada a uma infra-estrutura de ferramentas de teste do projeto.

O protótipo também é uma contribuição para o projeto QSEE – Qualidade de Software em Sistemas Embarcados[2]. Seu objetivo é transferir para a indústria brasileira do setor de software os conhecimentos adquiridos no INPE com o desenvolvimento de software para a área espacial, em particular os ambientes e técnicas de validação e verificação utilizadas na integração dos softwares embarcados em cargas úteis de missões de satélites científicos e balões estratosféricos. As dificuldades enfrentadas no estudo de caso do OBDH-EXPEmulator, como por exemplo, a necessidade de divisão das mensagens de acordo com seu tamanho, são informações relevantes quanto ao uso de algoritmos evolutivos para a geração automática de dados de teste que são pacotes de dados cujo tamanho é variável.

Em relação às contribuições à literatura da área, os primeiros resultados obtidos pela aplicação do GEO na geração automática de dados de testes foram apresentados em um artigo, publicado no 19^o Simpósio Brasileiro de Engenharia de Software, cuja referência é dada abaixo:

- B. T. Abreu, E. Martins, and F. L. Sousa. Automatic test data generation for path testing using a new stochastic algorithm. In *Proc. of the 19th Brazilian Symp. on Software Engineering*, volume 19, pages 247–262, Uberlândia, Brazil, 2005.

[1]Projeto aprovado pela FINEP, registrado com o número 1843/04.
[2]Projeto aprovado pela FINEP, registrado com o número 1960/04.

9.2 Sugestões para trabalhos futuros

A pesquisa desenvolvida nesta dissertação de mestrado aponta várias direções para trabalhos futuros. Talvez o mais interessante dentre eles é gerar dados de teste para o mesmo tipo de teste, mas utilizando outras funções-objetivo. O trabalho de Watkins [WH06] indica várias funções objetivo para o teste orientado a caminhos, e os resultados de suas comparações encorajam o uso de outras funções diferentes da *Similarity*. Não menos interessante que isto, outro trabalho futuro envolve o desenvolvimento de uma nova função objetivo para o critério de caminhos que combine as vantagens de outras funções, na tentativa de melhorar o resultado final.

Este trabalho avaliou a versão canônica do GEO e sua primeira variação, o GEO_{var}, para a geração de dados para cobrir caminhos no teste orientado a caminhos. Um trabalho futuro interessante é utilizar novas variações do GEO no mesmo problema. Além disso, há muitos outros requisitos de teste para os quais dados podem ser gerados utilizando o GEO; para isto, basta encontrar uma função objetivo que seja adequada ao requisito escolhido. O uso do GEO com sucesso para cobrir caminhos é um forte indicativo de que ele terá sucesso em outros requisitos, como definições e usos, por exemplo.

Um outro trabalho possível é utilizar reinicializações periódicas no GEO durante o processo de geração dos dados de teste. Sendo assim, caso a busca por dados não esteja progredindo, o processo de geração é reinicializado partindo de uma nova solução inicial.

Outro trabalho interessante é comparar o GEO para este tipo de problema não somente com algoritmos genéticos, mas também com outras metaheurísticas como, por exemplo, a Busca Tabu (em inglês, *Tabu Search*, TS) e o Recozimento Simulado (em inglês, *Simulated Annealing*, SA), que já foram utilizados para a geração automática de dados de teste.

Por fim, o uso do GEO para a geração de dados para teste caixa preta é outro ponto de investigação promissor. Neste caso, o GEO seria utilizado para gerar dados para testes baseados em modelos. Partindo dos documentos de especificação de requisitos de software, um modelo que represente o comportamento do software em teste é construído. Este modelo deve ser um modelo executável, uma vez que a abordagem de geração dinâmica de dados de teste requer que a execução do software em teste seja monitorada.

Referências Bibliográficas

[AC05] E. Alba and J. F. Chicano. Software Testing with Evolutionary Strategies. In N. Guelfi and A. Savidis, editors, *Rapid Integration of Software Engineering Techniques (RISE)*, volume 3943 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[AMS05] B. T. Abreu, E. Martins, and F. L. De Sousa. Automatic test data generation for path testing using a new stochastic algorithm. In *Proc. of the 19th Brazilian Symp. on Software Engineering*, volume 19, pages 247–262, Uberlândia, Brazil, 2005.

[Ang96] P. J. Angeline. Genetic programming's continued evolution. In Jr. K. E. Kinnear and P. J. Angeline, editors, *Advances in Genetic Programming 2*, chapter 1. MIT Press, 1996.

[Bak96] P. Bak. *How Nature Works: The Science of Self-Organized Criticality*. Springer, 1996.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[Bei95] B. Beizer. *Black-Box Testing*. John Wiley & Sons, 1995.

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT-A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, 1975.

[BFM97] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Institute of Physics Publishing and Oxford University Press, 1997.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1st edition, 1999.

[BJ02] P. M. S. Bueno and M. Jino. Automatic test data generation for program paths using genetic algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 12(6):691–710, 2002.

[BP01a] S. Boettcher and A. G. Percus. Extremal Optimization for Graph Partitioning. *Physical Review E*, 64:026114, 2001.

[BP01b] S. Boettcher and A. G. Percus. Optimization with Extremal Dynamics. *Physical Review Letters*, 86:5211–5214, 2001.

[Bre62] H. J. Bremermann. Optimization Through Evolution and Recombination. In M. C. Yovits, G. T. Jacobi, and G. D. Goldstine, editors, *Self-Organizing Systems*. Spartan Books, 1962.

[BS93] P. Bak and K. Sneppen. Punctuated Equilibrium and Criticality in a Simple Model of Evolution. *Physical Review Letters*, 71(24):4083–4086, 1993.

[BS02] H. Beyer and H. Schwefel. Evolution strategies - A comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002.

[BTW87] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality - An explanation of 1/f noise. *Physical Review Letters*, 59:381–384, July 1987.

[Bur03] I. Burnstein. *Practical Software Testing: A Process-oriented Approach*. Springer, 1st edition, 2003.

[Cha00] L. D. Chambers, editor. *The Practical Handbook of Genetic Algorithms*. CRC Press, 2000.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. on Software Engineering*, 2(3):215–222, September 1976.

[Coe03] L. S. Coelho. Fundamentos, Potencialidades e Aplicações de Algoritmos Evolutivos. In E. X. L. de Andrade, R. Sampaio, and G. N. Silva, editors, *Notas em Matemática Aplicada*. SBMAC, 2003.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[dLdC99] E. G. M. de Lacerda and A. C. P. L. F. de Carvalho. Introdução aos Algoritmos Genéticos. In C. Galvão and M. Valença, editors, *Sistemas Inteligentes - Aplicações a Recursos Hídricos e Ciências Ambientais*. Universidade Federal do Rio Grande do Sul Press, 1999.

[dMW01] A. R. C. da Rocha, J. C. Maldonado, and K. C. Weber. *Qualidade de Software: Teoria e Prática*. Prentice-Hall, 1st edition, 2001.

[dPE05] Instituto Nacional de Pesquisas Espaciais. EXPEmulator - Manual do Usuário 1.0 release 7. 2005.

[DTB03a] E. Díaz, J. Tuya, and R. Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *ASE*, pages 310–313, 2003.

[DTB03b] E. Díaz, J. Tuya, and R. Blanco. A modular tool for automated coverage in software testing. In *Proc. of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*, pages 241–246, Washington, DC, USA, 2003. IEEE Computer Society.

[Eld98] M. S. Eldred. Optimization strategies for complex engineering applications. Technical Report SAND–98-0340, Sandia National Labs, 1998. Disponível em `http://www.osti.gov/bridge/servlets/purl/642713-Vx0tNl/webviewable/642713.pdf`. Último acesso em 01/07/2006.

[Ele] ARC Electronics. Rs232 data interface: a tutorial on data interface and cables. Disponível em `http://www.arcelect.com/rs232.htm/`. Último acesso em 06/04/2006.

[ES03] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[EV03] M. C. F. P. Emer and S. R. Vergilio. Selection and Evaluation of Test Data Based on Genetic Programming. *Software Quality Journal*, 11(2):167–186, 2003.

[FA90] D. B. Fogel and W. Atmar. Comparing genetic operators with Gaussian mutation in simulated evolutionary processes using linear systems. *Biological Cybernetics*, 63:111–114, 1990.

[FDN59] R. M. Friedberg, B. Dunham, and J. H. North. A Learning Machine: Part II. *IBM Journal of Research and Development*, 3:282–287, 1959.

[FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transaction on Software Engineering Methodologies*, 5(1):63–86, 1996.

[Fog62] L. J. Fogel. Autonomous Automata. *Industrial Research*, 4:14–19, 1962.

[Fog90] D. B. Fogel. Simulated evolution: A 30-year perspective. In *Proc. of the Twenty-Fourth Asilomar Conference on Signals, Systems & Computers, Pacific Groove, CA*, volume 2, pages 1009–1014. Maple Press, 1990.

[Fri58] R. M. Friedberg. A Learning Machine: Part I. *IBM Journal of Research and Development*, 2:2–13, 1958.

[FV04] L. P. Ferreira and S. R. Vergilio. TDSGen: An Environment Based on Hybrid Genetic Algorithms for Generation of Test Data. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO'04) - Part II*, volume 3103, pages 1431–1432. Springer, 2004.

[GE77] S. J. Gould and N. Eldredge. Punctuated Equilibria: The Tempo and Mode of Evolution Reconsidered. *Paleobiology*, 3(2):115–151, 1977.

[GE93] S. J. Gould and N. Eldredge. Punctuated equilibrium comes of age. *Nature*, 366:223–227, 1993.

[GN97] M. J. Gallagher and V. L. Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[Gol89] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[GSR05a] R. L. Galski, F. L. De Sousa, and F. M. Ramos. Application of a New Multiobjective Evolutionary Algorithm to the Optimum Design of a Remote Sensing Satellite Constellation. In *Proc. of the 5th International Conference on Inverse Problems in Engineering: Theory and Practice*, Cambridge, UK, 11-15th July 2005.

[GSR05b] R. L. Galski, F. L. De Sousa, and F. M. Ramos. Discrete Optimal Design of Trusses by Generalized Extremal Optimization. In *Proc. of the 6th World Congresses of Structural and Multidisciplinary Optimization*, Rio de Janeiro, Brazil, 30th May - 03rd June 2005.

[GSRM04a] R. L. Galski, F. L. De Sousa, F. M. Ramos, and I. Muraoka. Spacecraft Thermal Design with the Generalized Extremal Optimization Algorithm. In *Proc. of the Inverse Problems, Design and Optimization Symposium, Rio de Janeiro, RJ, Brazil, 2004, (in CDROM)*, 2004.

[GSRM04b] R. L. Galski, F. L. De Sousa, F. M. Ramos, and I. Muraoka. Spacecraft thermal design with the generalized extremal optimization algorithm. In *Proc. of the Inverse Problems, Design and Optimization Symposium*, pages cd–rom, Rio de Janeiro, Brazil, 2004.

[HLV98] F. Herrera, M. Lozano, and J. L. Verdegay. Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis. *Artificial Intelligence Review*, 12(4):265–319, 1998.

[Hol62] J. H. Holland. Outline for a Logical Theory of Adaptive Systems. *Journal of the ACM*, 9(3):297–314, 1962.

[Hol75] J. H. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.

[How77] W. E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. on Software Engineering*, SE-3(4):266–278, 1977.

[HSK+90] G. A. Held, D. H. Solina, D. T. Keane, W. J. Haag, P. M. Horn, and G. Grinstein. Experimental Study of Critical-Mass Fluctuations in an Evolving Sandpile. *Physical Review Letters*, 65(9):1120–1123, 1990.

[INP] Instituto nacional de pesquisas espaciais. `http://www.inpe.br/`. Último acesso em 04/04/2006.

[JMF98] V. A. Santiago Jr. and M. F. Matiello-Francisco. EXP-OBDH Communications protocol definition: a case study for PLAVIS. Technical report, Instituto Nacional de Pesquisas Espaciais, 1998.

[Jon99] K. De Jong. Genetic Algorithms: A 30 Year Perspective, 1999. Disponível em `http://www.pscs.umich.edu/jhhfest/Papers/dejong.ps.gz`. Último acesso em 01/09/2005.

[JSE96] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[KBAK99] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, 1999.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KJV83] S. Kirkpatrick, D. Gelatt Jr., and M. P. Vecchi. Optimization by Simmulated Annealing. *Science*, 220(4598):671–680, 1983.

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Trans. on Software Engineering*, 16(8):870–879, 1990.

[Koz89] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proc. of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, San Mateo, CA, USA*, volume 1, pages 768–774. Morgan Kaufmann, 1989.

[Koz90] J. R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990. Disponível em `http://www.genetic-programming.com/jkpdf/tr1314.pdf`. Último acesso em 01/09/2005.

[Koz92] J. R. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, 1992.

[KP03] J. R. Koza and R. Poli. A Genetic Programming Tutorial. In E. Burke, editor, *Introductory Tutorials in Optimization, Search and Decision Support*, chapter 8. 2003. Disponível em `http://www.genetic-programming.com/jkpdf/burke2003tutorial.pdf`. Último acesso em 01/09/2005.

[LY01] J. Lin and P. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences*, 131(1-4):47–64, 2001.

[McC76] T. J. McCabe. A complexity measure. In *ICSE '76: Proc. of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[McM04] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[Met] Metrics. `http://metrics.sourceforge.net/`. Último acesso em 02/03/2006.

[Meu01] C. Meudec. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification & Reliability*, 11(2):81–96, 2001.

[MH03] P. McMinn and M. Holcombe. The State Problem for Evolutionary Testing. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO2003), Chicago, IL, USA, July 12-16*, volume 2724 of *Lecture Notes in Computer Science*, pages 2488–2498. Springer, 2003.

[Mic92] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 2nd edition, 1992.

[MMS01] C. C. Michael, G. McGraw, and M. Schatz. Generating Software Test Data by Evolution. *IEEE Trans. on Software Engineering*, 27(12):1085–1110, 2001.

[MS76] W. Miller and D. L. Spooner. Automatic Generation of Floating-Point Test Data. *IEEE Trans. on Software Engineering*, 2(3):223–226, 1976.

[MS04] N. Mansour and M. Salame. Data Generation for Path Testing. *Software Quality Journal*, 12(2):121–136, 2004.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, 1979.

[Oul91] M. A. Ould. Testing: a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, 1991.

[PGGZ94] M. Pei, E. Goodman, Z. Gao, and K. Zhong. Automated Software Test Data Generation Using A Genetic Algorithm. Technical Report 6/2/1994, Michigan State University, 1994. Disponível em `http://www.egr.msu.edu/~pei/paper/GApaper94-02.ps`. Último acesso em 28/03/2006.

[PHP99] R. P. Pargas, M. J. Harrold, and R. P. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[Pre97] R. S. Pressman. *Software Engineering: A practitioner's approach*. McGraw-Hill, 4th edition, 1997.

[Rau86] D. M. Raup. Biological extinction in earth history. *Science*, 213(4745):1528–1533, 1986.

[SGJ92] A. C. Schultz, J. J. Grefenstette, and K. A. De Jong. Adaptive testing of controllers for autonomous vehicles. In *Proc. of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV'92)*, pages 158–164, Washington, DC, USA, 1992. IEEE Computer Society.

[SHR01] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. *ACM Transactions on Software Engineering Methodologies*, 10(2):209–254, 2001.

[Som03] I. Sommerville. *Engenharia de Software*. Prentice-Hall, 6th edition, 2003.

[Sou02] F. L. De Sousa. *Otimização Extrema Generalizada: Um novo algoritmo estocástico para o projeto ótimo*. PhD thesis, Instituto Nacional de Pesquisas Espaciais (INPE), 2002. INPE-9564-TDI/836.

[SR02] F. L. De Sousa and F. M. Ramos. Function Optimization Using Extremal Dynamics. In *Proc. of the 4th International Conference on Inverse Problems in Engineering*, volume I, pages 115–119, Rio de Janeiro, Brazil, 2002.

[SRPG03] F. L. De Sousa, F. M. Ramos, P. Paglione, and R. M. Girardi. New Stochastic Algorithm for Design Optimization. *AIAA Journal*, 41(9):1808–1818, September 2003.

[SRSS05] F. L. De Sousa, F. M. Ramos, F. J. C. P. Soeiro, and A. J. Silva Neto. Application of the Generalized Extremal Optimization Algorithm to an Inverse Radiative Transfer Problem. In *Proc. of the 5th International Conference on Inverse Problems in Engineering: Theory and Practice*, Cambridge, UK, 11-15th July 2005.

[ST05] F. L. De Sousa and W. K. Takahashi. Generalized Extremal Optimization Applied to Three-Dimensional Truss Design. In *Proc. of the 18th International Congress of Mechanical Engineering (COBEM2005)*, Ouro Preto, Brazil, 2005.

[Sth96] H. Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[SVR04] F. L. De Sousa, V. Vlassov, and F. M. Ramos. Generalized Extremal Optimization: An application in Heat Pipe Design. *Applied Mathematical Modeling*, 28:911–931, 2004.

[TCM98] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. *SIGSOFT Softw. Eng. Notes*, 23(2):73–81, 1998.

[Tra97] N. Tracey. Test-case data generation using optimisation techniques – first year DPhil report. Department of Computer Science, University of York,

1997. Disponível em `http://www.cs.york.ac.uk/testsig/publications/njt-jun97.pdf`. Último acesso em 03/03/2006.

[Tur92] A. M. Turing. Intelligent machinery. In D. C. Ince, editor, *Mechanical Intelligence: Collected Works of A. M. Turing*. North-Holand, 1992.

[Van98] G. N. Vanderplaats. *Numerical optimization techniques for engineering design*. Colorado Springs: Vanderplaats Research & Development, 2nd edition, 1998.

[WH06] A. Watkins and E. M. Hufnagel. Evolutionary test data generation: a comparison of fitness functions. *Software Practice and Experience*, 36(1):95–116, 2006.

[Whi01] D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001. Disponível em `http://www.cs.colostate.edu/sched/pubs/overview.ps.gz`. Último acesso em 01/09/2005.

[Wik06a] Wikipedia. Charles Darwin. 2006. Disponível em `http://en.wikipedia.org/wiki/Charles_Darwin`. Último acesso em 22/02/2006.

[Wik06b] Wikipedia. Computer Bugs. 2006. Disponível em `http://en.wikipedia.org/wiki/Computer_bugs`. Último acesso em 28/03/2006.

[Wik06c] Wikipedia. Control Flow. 2006. Disponível em `http://en.wikipedia.org/wiki/Control_flow`. Último acesso em 20/06/2006.

[Wik06d] Wikipedia. Metaheuristic. 2006. Disponível em `http://en.wikipedia.org/wiki/Metaheuristic`. Último acesso em 01/07/2006.

[WM97] D. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Trans. on Evolutionary Computation*, 1(1):67–82, 1997.

[WW06] A. Wollrath and J. Waldo. Trail: RMI. 2006. Disponível em `http://java.sun.com/docs/books/tutorial/rmi/index.html`. Último acesso em 08/06/2006.