Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e de Computação
Departamento de Engenharia de Computação e Automação Industrial

Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software

Paulo Marcos Siqueira Bueno

Orientador: Prof. Dr. Mario Jino

Campinas - SP Brasil
1999

Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Departamento de Engenharia de Computação e Automação Industrial

Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software

Paulo Marcos Siqueira Bueno

Orientador: Prof. Dr. Mario Jino

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação (FEEC) da Universidade Estadual de Campinas (Unicamp), como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica e Computação.

Este trabalho contou com o apoio financeiro da FAPESP — Fundação de Amparo à Pesquisa do Estado de São Paulo, processo 97/07004-7.

Campinas - SP Brasil

1999

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

B862g

Bueno, Paulo Marcos Siqueira
Geração automática de dados e tratamento de não executabilidade no teste estrutural de software. / Paulo Marcos Siqueira Bueno.--Campinas, SP: [s.n.], 1999.

Orientador: Mario Jino
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1.Software - Testes. 2. Algoritmos genéticos. 3. Engenharia de software. 4. Software – Validação. I. Jino, Mário. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e de Computação

Departamento de Engenharia de Computação e Automação Industrial

Geração Automática de Dados e Tratamento de Não Executabilidade no Teste Estrutural de Software

Paulo Marcos Siqueira Bueno

Dissertação de Mestrado defendida e aprovada, em 02 de Junho de 1999, pela Banca Examinadora constituída pelos professores:

Prof. Dr. Mario Jino, Presidente
DCA/FEEC/Unicamp

Prof. Dr. Ivan Luiz Marques Ricarte
DCA/FEEC/Unicamp

Prof. Dr. José Carlos Maldonado
ICMSC/USP - São Carlos

Aventurar-se causa ansiedade, mas deixar de arriscar-se é perder a si mesmo. E aventurar-se no sentido mais elevado é precisamente tomar consiência de si próprio.

Kierkgaard

À minha mãe e à memória de meu pai.

Resumo

Este trabalho apresenta uma ferramenta e técnicas para automação da geração de dados e identificação de não executabilidade para a técnica estrutural de teste de software. A ferramenta baseia-se na *Técnica Dinâmica*; na busca utilizando *Algoritmos Genéticos*; e no reuso de soluções passadas através do *Raciocínio Baseado em Casos*. O objetivo é gerar automaticamente dados de entrada que executem caminhos completos no programa em teste e identificar a não executabilidade desses caminhos quando for o caso; o que é feito através da *Heurística de Identificação Dinâmica de Potencial não Executabilidade* proposta. Um experimento conduzido mostra a validade das soluções elaboradas e do benefício da utilização da ferramenta. Os resultados alcançados sugerem que, apesar da indecidibilidade geral desses problemas, soluções parciais podem ser úteis à prática do teste de software.

Abstract

A tool and techniques are presented for test data generation and infeasibility identification in structural software testing technique. The tool is based on: the *Dynamic Technique*; searching using *Genetic Algorithms*; and reusing of solutions through *Case-Based Reasoning*. The objective is to automatically generate input data which execute complete paths in a program and identify path infeasibility when this is the case; this is done through the *Potential Infeasibility Dynamic Identification Heuristic* proposed. An experiment shows the validity of the developed solutions and the benefit of using the tool. Results attained indicate that, despite the general undecidability of the problems, partial solutions may be useful to software testing practice.

Agradecimentos

Agradeço muito ao Professor Mario Jino pela orientação essencial ao desenvolvimento deste trabalho, pelo aprendizado propiciado e pelo convívio enriquecedor nos aspectos profissional e pessoal.

Ao Professor José Carlos Maldonado pelas sugestões feitas ao trabalho e pela motivação durante o projeto PVTSO - Telebrás e nos contatos mantidos no período do mestrado.

Ao Professor Ivan Luiz Marques Ricarte pelo estímulo e pelas contribuições feitas ao trabalho.

Ao pessoal do Grupo de Testes DCA/FEEC/Unicamp: Adalberto Crespo, Cláudia Tambascia, Maria Cristina, Daniela Cruzes, Inês Boaventura, Ivan Granja, Letícia Peres, Nélson Mendes e Plínio Vilela pela amizade, colaboração e pelas críticas essenciais ao aprimoramento das idéias. Em especial agradeço à Silvia Vergílio pela grande ajuda nas explorações iniciais do tema, pelo incentivo e prestatividade constantes; ao Marcos Chaim pelas explicações sobre a implementação da POKE-TOOL e pelas diversas sugestões; e ao Edmundo Spoto pelas consultas matemáticas e pela força em todos momentos.

Ao Maurício Figueredo e ao Leandro Nunes pelas discussões sobre Algoritmos Genéticos; ao Ronaldo Camilo pela conversa inicial sobre Raciocínio Baseado em Casos e ao Marcelo Malheiros pelas dicas sobre UNIX. À Viviane Petito pelo competente apoio técnico.

A todos os amigos do L.C.A. pela companhia diária e pelo companherismo.

Aos amigos: Rosana Spoto, Fabiano, Maria Adela e Renato Tutumi, Lucimara Desiderá, Raquel Schulze e Bruno Schulze, Luciano Krob, Ana Cristina Melo, Hermelinda e Marquinho pelos muitos bons momentos nos últimos anos... . Ao pessoal de Belo Horizonte: Sérgio Paiva e Mônica, Luciana Otávia, Glória Regina e Fernando Monteiro pela amizade de sempre.

À minha família: Lígia, Luisa, Matheus, Lucas, Rachel, Débora, Ailton e Samuel pelo apoio em todos os sentidos. Aos meus pais por terem proporcionado o ambiente favorável à minha formação, em especial à minha mãe Edna por tudo de positivo que ela representa e pela torcida fundamental.

À Luciana pelo grande carinho, pela ajuda e compreensão nos momentos de desânimo e por ter compartilhado comigo a satisfação decorrente da realização deste trabalho.

À FAPESP pelo apoio financeiro.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capítulo o trabalho é situado em seu contexto; é destacada a motivação para a realização do mesmo e são identificados os objetivos que delinearam os estudos realizados tendo em vista o desenvolvimento do tema proposto. É mostrada também a organização desta dissertação.

1.1 Contexto

Houve nos últimos anos um grande desenvolvimento tecnológico em relação aos recursos de hardware disponíveis. Este desenvolvimento foi acompanhado de um crescimento de demanda por software cada vez mais complexo e de caráter abrangente.

Este cenário tem como conseqüência uma busca por software diversificado, muitas vezes complexo e, em alguns casos, com altos requisitos de confiabilidade. Neste contexto, a Engenharia de Software tem contribuído para aumentar a produção de software e, sobretudo, a sua qualidade, através da proposição de métodos, ferramentas e procedimentos cientificamente elaborados.

Idealmente, durante o processo de desenvolvimento devem ser utilizadas técnicas e metodologias a fim de tornar este processo o mais confiável possível. No entanto, por ser uma atividade fortemente baseada no trabalho humano, o desenvolvimento de software está sujeito à inerente incapacidade humana de agir e comunicar-se com perfeição.

Uma atividade de grande importância para o aumento da qualidade do software é o teste. Esta atividade representa a última possibilidade revelar erros na especificação, no projeto e na codificação, antes que o produto seja posto efetivamente em funcionamento. Conforme a definição de Myers [MYE79] "Teste é o processo de executar o programa com o objetivo de encontrar defeitos". Segundo destaca Beizer [Bei95], indiretamente, o teste pode fornecer informações para o aprimoramento do processo de desenvolvimento.

Uma etapa importante do teste consiste na seleção dos dados de entrada, isto é, dos valores a serem utilizados para a execução do programa. Esta seleção é baseada em critérios que visam a identificar "bons" dados de teste, isto é, dados que possuam uma elevada probablilidade de revelar defeitos no programa.

São considerados neste trabalho os critérios de teste *Potenciais Usos* [Mal91]. Tais critérios têm enfoque estrutural e são baseados em análise de fluxo de dados, isto é, os requisitos de teste selecionados na aplicação do critério ao programa refletem o fluxo de controle e o fluxo de dados do mesmo.

O teste com esses critérios é apoiado pela ferramenta de teste *POKE-TOOL* [Cha91, Mal91]. Através desta ferramenta é possível realizar a análise estática do código fonte, gerando os requisitos de teste desses critérios e um programa instrumentado, que permite o monitoramento da execução dos dados selecionados. É possível também avaliar (após a aplicação dos dados selecionados) os resultados do teste, ou seja, a proporção de requisitos de teste efetivamente exercitados por esses dados.

Os requisitos gerados pela ferramenta (referidos também como elementos requeridos) direcionam a seleção de dados de teste; entretanto, cabe ao testador encontrar conjuntos de valores de entrada para o programa tais que os requisitos de teste sejam satisfeitos.

Dado este contexto pode-se identificar a motivação para a proposição do tema e estabelecer os objetivos deste trabalho.

1.2 Motivação e Objetivos

Segundo Pressman [Pre97] a atividade de teste consome tipicamente entre 40% e 50% do esforço total no desenvolvimento de Software. Um aspecto essencial para a redução deste custo é a utilização de ferramentas automatizadas que apoiem esta atividade.

Uma tarefa não automatizada no processo de teste utilizando a *POKE-TOOL* é a geração de dados de entrada que exercitem os elementos requeridos pelos critérios. O testador deve selecionar esses dados através da análise do código fonte. Esta é uma tarefa extremamente complexa que requer esforço mental, conhecimentos de aspectos conceituais do critério utilizado e do código do programa em tese.

A automação da atividade de geração de dados de teste é altamente desejável; entretanto, esta automação é inviável em casos gerais. Conforme destaca Clarke [Cla76], este problema é equivalente ao "problema da parada", reconhecidamente indecidível. A questão da não executabilidade — existência de elementos requeridos não executáveis — também impede a automação completa desta atividade [FW88].

Devido a essas limitações, trabalhos que abordam a automação da geração de dados de teste, assim como os que tratam da questão da não executabilidade, procuram obter

soluções parciais, aplicáveis a classes particulares de programas.

Tendo em vista essas considerações foram definidos os objetivos deste trabalho:

- Realizar estudos sobre a automação da geração de dados de teste e não executabilidade no teste estrutural;
- Definir uma abordagem para o tratamento desses problemas no contexto da ferramenta *POKE-TOOL* para a aplicação dos critérios de teste *Potenciais Usos*;
- Definir os requisitos, desenvolver e implementar uma ferramenta que auxilie o testador na tarefa de gerar dados de teste e de identificar caminhos não executáveis;
- Validar a ferramenta através de experimento e análise de casos.

A organização desta dissertação é descrita a seguir.

1.3 Organização

O **Capítulo 2** discute conceitos relacionados ao teste de software. É enfatizada a *técnica estrutural*; são apresentadas definições básicas e são introduzidos conceitos de critérios relacionados a esta técnica.

O **Capítulo 3** aborda o problema da automação da geração de dados de teste. São identificados objetivos, pontos de difícil tratamento e abordagens existentes na literatura. É situado também o contexto no qual a ferramenta aqui proposta se insere.

O **Capítulo 4** descreve a solução desenvolvida. Primeiramente é fornecida uma visão geral da ferramenta proposta. É introduzida a busca utilizando Algoritmos Genéticos (A.G.'s) e é descrita a utilização desses algoritmos neste trabalho. É apresentado o modelo de instrumentação desenvolvido e a idéia de reuso de soluções baseado em analogia. É apresentado e analisado o tratamento dinâmico para a questão da não executabilidade.

O **Capítulo 5** trata da validação da ferramenta e da análise das idéias desenvolvidas e implementadas. É definido o objetivo da validação e é descrito o experimento conduzido. São apresentados e comentados os resultados obtidos nos programas utilizados, incluindo a análise de situações particulares. Os resultados globais são sumarizados e analisados.

O **Capítulo 6** apresenta as conclusões deste trabalho e indica possíveis extensões.

Capítulo 2

Teste de Software

O desenvolvimento de sistemas de software envolve uma série de atividades fortemente baseadas no trabalho humano. Devido à incapacidade humana de agir e comunicar-se com perfeição o desenvolvimento do software deve ser acompanhado por atividades de garantia de qualidade. Segundo Beizer, erros ocorridos no desenvolvimento persistem como "bugs" no software. Esses erros são causados pela capacidade limitada de humanos de tratar problemas complexos e de comunicar-se de forma perfeita, sem equívocos de interpretação [Bei95].

O Teste de Software é um procedimento crítico para a qualidade do software, representando a última atividade na qual é possível revelar erros na especificação, no projeto e na codificação do software [Pre97].

Este capítulo aborda a atividade teste de software: é destacada a idéia básica do teste; são descritos os objetivos desta atividade; são definidos conceitos importantes, com ênfase na técnica estrutural de teste. São abordados os critérios de teste estrutural com destaque para os critérios Potenciais Usos [Mal91].

2.1 Idéia Básica

A *idéia básica* do teste é executar um programa, fornecendo dados de entrada (referidos como *dados de teste*) e comparar os valores de saída produzidos com o resultado esperado, segundo a especificação do programa. Caso a saída alcançada seja diferente do resultado esperado tem-se uma *falha* do programa. A causa desta discrepância deve ser identificada e corrigida no processo de depuração [1].

[1] A depuração não é teste. Na depuração busca-se relacionar sintomas observados no teste (saídas incorretas do programa) com as suas causas (defeitos no programa). O objetivo é a identificação e correção desses defeitos.

Informalmente *dado de entrada* e *dado de teste* referem-se ao conjunto de valores utilizados como entrada para o programa em uma determinada execução. Um *caso de teste* consiste de um dado de teste associado a um resultado esperado do processamento, segundo a especificação do software.

Uma questão importante é a identificação da ocorrência de uma falha. No teste pressupõe-se a existência de um mecanismo (automatizado ou manual) para determinar se os valores de saída são corretos. Este mecanismo — um *oráculo* — deve confrontar os valores de saída obtidos com os que seriam esperados do programa, segundo a especificação (formal ou informal) do mesmo. A existência de um oráculo — comumente o testador — é essencial para que o processo de teste faça sentido [Wey82] [2].

2.2 Objetivos do Teste

"Teste é o processo de executar o programa com o objetivo de encontrar defeitos" [MYE79]. Neste sentido, um bom caso de teste deve possuir uma grande probablilidade de revelar um defeito ainda não descoberto. Um benefício secundário do teste é dar indícios de que o software está se comportando de acordo com as especificações. Entretanto, o teste não pode mostrar a ausência de defeitos, apenas a presença desses. "Nós tentamos *quebrar* ("break") o software porque este é o único meio prático pelo qual é possível estabelecer confiança de que o produto está adequado para o uso" [Bei95].

Segundo Hetzel [Het73] teste é "o processo de estabelecimento de confiança de que um programa ou sistema faz o que supõe-se que ele faça".

Beizer [Bei95] enfatiza a importância do teste para o aprimoramento da qualidade do software. Neste sentido o teste pode ser visto como "um processo de ganho de informação sobre o software a qual, quando realimentada aos programadores, permite evitar enganos passados e aprimorar o software futuro".

Portanto, existem dois enfoques essenciais e complementares: o teste como uma atividade de validação das funcionalidades que o software deve apresentar, procurando demonstrar o comportamento correto — chamado de "clean test" [Bei95]; e o teste feito com o propósito de "quebrar" o software, chamado de "dirty test". Segundo [MYE79] e [Gra94], considerar o primeiro enfoque apenas tende a resultar em um teste de baixa qualidade. Isto é, além de verificar se o software faz o que é esperado que ele faça é necessário também verificar se este software não faz — erroneamente — o que **não** é esperado que ele faça.

Tem-se deste modo que a confiança na corretitude do software é construída através da

[2] O oráculo não decide se os valores foram computados corretamente, apenas se eles são corretos. Também não identifica o defeito no programa e não fornece os valores de saída corretos.

tentativa sistemática de revelar defeitos nele presentes [3].

Trata-se de uma atividade "destrutiva" do ponto de vista psicológico, no sentido de que existe o objetivo de revelar os problemas existentes no produto, recém elaborado, utilizando tarefas "construtivas" relacionadas à engenharia de software.

2.3 Plano, Estratégia e Atividades de Teste

O teste deve estar associado a um *plano de teste*. O plano contempla os objetivos do teste, recursos a serem utilizados, estratégia a ser seguida, técnicas e critérios a serem aplicados, tempo disponível, critério para a avaliação de completude do teste, dentre outros aspectos.

Uma *estratégia* sistemática para o teste consiste em estabelecer passos que correspondam às etapas do processo de engenharia de software [4]. Deste modo o teste é abordado em quatro *etapas*, com enfoques distintos: 1) *teste de unidade*: cada unidade implementada é testada para identificação de deficiências algorítmicas e nas estruturas de dados; 2) *teste de integração*: baseado no projeto e na arquitetura do software, visa a revelar problemas na relação e interface entre os módulos; 3) *teste de validação*: validação dos requisitos de software estabelecidos na fase de análise; 4) *teste de sistema*: teste do software com outros elementos do sistema [Pre97].

Em todas as etapas o teste está associado a uma série de *atividades*: planejamento, projeto de casos de teste, execução de casos de teste e análise dos resultados.

2.4 Engano, Defeito, Erro e Falha

O entendimento desses termos é essencial para a correta interpretação de trabalhos nesta área. Em [IEE90] têm-se as seguintes definições:

Uma *falha* ("failure") acontece quando uma saída incorreta é produzida com relação a especificação do programa. Um *defeito* ("bug") é uma deficiência no programa que pode provocar uma saída incorreta para algum dado de entrada. Um *erro* ("error") é uma diferença existente entre o valor correto e o valor esperado para alguma variável na execução do programa, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa, constitui um erro. Um *engano* ("mistake") é uma ação humana que produz um resultado incorreto, por exemplo, um comando incorreto em um programa.

[3] Este fato é referido como *paradoxo do teste*: Encontrar defeitos "destroi" a confiança no software e o propósito do teste é justamente construí-la. O ponto é que a tentativa sem sucesso de "destruir" o software permite a construção da confiança no mesmo [Gra94].

[4] Engenharia de sistema, análise de requisitos de software, projeto e implementação [Pre97].

Deste modo: um engano do programador pode provocar a inserção de um defeito no software (Exemplo: trocar o comando $x = 2^y$ por $x = 2 \times y$). Este defeito, quando exercitado por um dado de entrada adequado, pode produzir um erro (No exemplo $y = 3$ produz um erro, enquanto $y = 2$ não o produz, pois neste caso $2^y = 2 \times y$). Este erro pode produzir uma falha se ele manifestar-se na saída do programa (No exemplo, se $y = 3$ e x for uma saída do programa, ocorreria uma falha) [5].

2.5 Técnicas e Critérios de Teste

O teste de todas as combinações possíveis de valores de entrada para um programa não é factível, pois o número de combinações tende a ser exorbitante. Neste fato reside a inviabilidade de, através de uma abordagem geral de teste, provar a corretude de um programa [Ham98]. Torna-se necessário, portanto, selecionar um sub-conjunto de dados de teste a serem utilizados. Neste sentido foram definidas diversas *técnicas* que fornecem uma abordagem sistemática para o teste.

As diversas técnicas de teste utilizam diferentes tipos de informação visando escolher bons dados de teste, que aumentem a probabilidade da revelação de defeitos. Vários *Critérios de Teste*, associados às técnicas, têm sido propostos com este objetivo. Essencialmente os *Critérios de Teste* estabelecem requisitos a serem satisfeitos e que podem orientar: a seleção de dados de teste; a avaliação da qualidade do teste e a definição de requisitos de suficiência a serem atingidos para o encerramento desta atividade.

Satisfazer um critério de teste, de um modo genérico, significa satisfazer todos os requisitos de teste estabelecidos pelo critério.

Podem ser caracterizadas três técnicas principais, às quais estão relacionados diversos Critérios de Teste.

Técnica Baseada em Defeitos: Os dados de teste são gerados baseando-se em classes de defeitos específicos comuns em programas. O objetivo é mostrar a presença ou ausência de tais defeitos no programa. Um exemplo é a "Análise de Mutantes" [DMLS78, Del93]. Geram-se "programas mutantes" com alterações em relação ao programa em teste. Deve ser selecionado um conjunto de dados de entrada a fim de distinguir o comportamento do programa original do comportamento de cada um dos programas mutantes.

Técnica Funcional: Estabelece casos de teste baseados na especificação e na identificação dos requisitos funcionais. Examina aspectos fundamentais da funcionalidade do software. Exemplos: Análise de Valor Limite, Particionamento de Equivalência [Pre97] e

[5] Esta é apenas uma ilustração dos conceitos. A ocorrência de uma falha do software pode ser provocada por problemas no sistema como um todo, não necessariamente por um defeito do software. Neste sentido, a confiabilidade do software é necessária, mas não suficiente, para a confiabilidade do sistema.

Grafos de Causa-Efeito [MYE79].

Técnica Estrutural: É baseada na estrutura da implementação, utilizando o código fonte para identificar requisitos a serem satisfeitos pelos casos de teste [Pre97]. Critérios de teste estrutural estabelecem componentes estruturais do programa a serem exercitados pelo testador.

As diversas técnicas não são excludentes. Isto é, não se deve utilizar apenas uma ou outra técnica: elas são complementares, pois cada uma delas testa o software sob uma perspectiva diferente. Geralmente, uma técnica proporciona a descoberta de classes de defeitos diferentes das que as outras proporcionam [Mal91] .

No contexto desta dissertação. são de interesse critérios de teste relacionados à técnica estrutural. Na seção seguinte são apresentadas definições básicas essenciais ao entendimento deste trabalho.

2.6 Técnica Estrutural de Teste

Discutem-se nesta seção conceitos associados à técnica estrutural de teste. São feitas definições importantes e discutidos critérios de teste associados a esta técnica.

2.6.1 Definições Básicas

Serão discutidas a seguir algumas definições importantes associadas ao teste estrutural. A definição de critérios de teste requer a formalização de vários conceitos. Procurou-se estabelecer um sub-conjunto desses conceitos essencial ao problema da geração automática de dados de teste, tratado neste trabalho. Exceto quando são feitas referências explícitas utilizaram-se como base os trabalhos de Maldonado [Mal91] e Korel [Kor90].

Utiliza-se genericamente o termo *programa* para referir-se a um procedimento ou função escrito em alguma linguagem de programação. *Unidade* e *Módulo* são termos freqüentemente usados com o mesmo sentido.

Um programa P pode ser decomposto em um conjunto de *blocos* disjuntos de comandos; a execução do primeiro comando do bloco acarreta a execução de todos os outros, na ordem dada, desse bloco. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor; e exatamente um único sucessor, exceto possivelmente o último comando.

A estrutura do programa P pode ser representada por um grafo dirigido com um único nó de entrada e um único nó de saída. Neste grafo *nós* correspondem aos blocos e representam-se possíveis fluxos de controle entre blocos através de *arcos*.

Portanto, a estrutura de P pode ser representada por um grafo dirigido $G = (N, E, s, e)$, onde N é um conjunto de nós, E um conjunto de arcos, s é um único nó de entrada e e um único nó de saída. Um arco (n_i, n_j) corresponde a uma possível transferência de controle entre os nós i e j. Um arco (n_i, n_j) é chamado de *ramo* se o último comando de n_i é um comando de seleção ou de repetição. A cada *ramo* pode ser associado um predicado denominado *predicado de ramo* [6].

Um *sub-caminho* é uma seqüência de nós $(n_1, n_2, ..., n_k)$, $k \geq 2$, tal que existe um arco de n_i para n_{i+1} (isto é, $(n_i, n_{i+1}) \in E$).

Um *caminho* ou *caminho completo* é um sub-caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G [7]. Além disso, um *caminho pretendido* é um caminho o qual deseja-se executar.

As ocorrências de uma variável em um programa podem ser uma *definição de variável*, um *uso de variável* ou uma *indefinição*.

Uma *definição de variável* ocorre quando um valor é armazenado em uma posição de memória; em geral, isto ocorre em um programa quando a variável está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; iii) em chamadas de procedimento como parâmetro de saída, neste último caso tem-se uma *definição por referência*.

A ocorrência de uma variável é um *uso* quando a referência a essa variável não a estiver definindo. Podem ser distinguidos dois tipos de uso. O *c-uso* afeta diretamente a computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado. O *p-uso* afeta diretamente o fluxo de controle do programa.

Uma variável está *indefinida* quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar definida na memória.

Um caminho $(i, n_1, ..., n_m, j)$, $m \geq 0$, que não contenha definição de uma variável x nos nós $n_1, ..., n_m$ é chamado de *caminho livre de definição com respeito a* x do nó i ao nó j e do nó i ao arco (n_m, j).

Uma *variável de entrada* x_i para o programa P é uma variável que aparece em um comando de entrada; um parâmetro de entrada; ou ainda, uma variável de escopo global utilizada no programa. Variáveis de entrada podem ser dos diferentes tipos tratados pela linguagem.

$I = (x_1, x_2, ..., x_n)$ é o vetor de variáveis de entrada do programa P.

---

[6] Em diversos trabalhos os termos *arcos* e *ramos* são utilizadados como sinônimos. Adotou-se neste trabalho a nomeclatura definida por Korel [Kor90], na qual *ramos* são *arcos* aos quais estão associados predicados.

[7] Neste trabalho utilizam-se os termos *caminho* e *caminho completo* como sinônimos. Sempre que for necessário fazer menção a um caminho "não completo" será utilizado o termo *sub-caminho*.

O *domínio* D_{xi} da variável de entrada x_i é o conjunto de todos os valores que x_i pode assumir.

O *domínio de entrada* D do programa P é o produto cartesiano $D = D_{x1} \times D_{x2} \times ... \times D_{xn}$, onde D_{xi} é o domínio da variável de entrada x_i.

Um único ponto x no espaço de entrada n-dimensional D, $x \in D$ é referido como um *dado de entrada* ou *dado de teste* [8].

Um caminho é *executável* se existe algum dado de entrada $x \in D$ para o qual o caminho é atravessado durante a execução do programa, caso contrário, tal caminho é *não executável.*

2.6.2 Critérios de Teste Estrutural

Critérios de teste estrutural estabelecem componentes estruturais do programa a serem exercitados. Satisfazer um critério de teste estrutural significa exercitar todos os componentes requeridos pelo critério.

Inicialmente tais critérios foram baseados no fluxo de controle de programas. Pode-se citar como exemplo o critério *todos os nós*, o critério *todos os ramos*, e o *critério todos os caminhos* que exigem, respectivamente, que cada nó, que cada ramo e que cada caminho do grafo seja executado pelo menos uma vez. A idéia intuitiva associada a esses critérios é que não se pode confiar em um programa que não teve cada um de seus comandos executados pelo menos uma vez.

Os critérios todos os nós e todos os ramos são aplicáveis, por requererem uma quantidade finita de dados de teste para serem satisfeitos; entretanto, por não requererem combinações de execução dos comandos, podem deixar de revelar a presença de defeitos simples. Por outro lado, o teste de todos os caminhos é impraticável devido ao seu custo.

Os *critérios baseados em análise de fluxo de dados* usam informações do fluxo de dados do programa para derivar os requisitos de teste. A idéia intuitiva neste caso é que não se deve considerar suficientemente testado um programa se todos os resultados computacionas não tiverem sido usados pelo menos uma vez [RW85]. Essencialmente, tais critérios exigem a execução de caminhos do ponto onde uma variável foi definida, até o ponto onde ela foi utilizada (ou potencialmente utilizada). A razão principal da proposição desses critérios é estabelecer uma ponte entre o critérios todos os ramos e o critério todos os caminhos.

Diversos autores definiram famílias de critérios baseados em análise de fluxo de dados:

Família de Critérios Baseados em Fluxo de Dados de Rapps e Weyuker [RW85], Família de Critérios Potenciais Usos de Maldonado, Jino e Chaim [MCJ88, Mal91], Família de

[8] Este conceito já foi definido informalmente no início deste capítulo.

Critérios K-tuplas requeridas de Ntafos [Nta84], Família de Critérios Baseada em Contexto de Dados de Laski e Korel [LK83], Critérios Restritos de Teste de Software de Vergilio, Jino e Maldonado [Ver97].

Uma informação importante associada à utilização de critérios é o *nível de cobertura atingido* no teste. Esta informação relaciona-se à utilização de critérios como um meio para a avaliação da qualidade do teste. O nível de cobertura determina o percentual de elementos requeridos pelo critério efetivamente exercitados pelo conjunto de dados de teste aplicados. Deste modo, quanto maior este valor, mais completo terá sido o teste [9].

Família de Critérios Potenciais Usos

Neste trabalho é de particular interesse a família de critérios *Potenciais Usos* [Mal91]. A idéia básica desta família de critérios consiste em requerer associações de fluxo de dados independentemente da ocorrência explícita de uma referência a uma determinada definição de variável: se o uso desta definição pode existir, a potencial associação é requerida. Em outras palavras, requerem basicamente que caminhos livres de definição, em relação a qualquer nó i, que possua definição de variável e a qualquer variável x definida em i, sejam executados, independentemente de ocorrer uso dessa variável nesses caminhos.

Deste modo abre-se a possibilidade do critério levar o testador a selecionar dados de teste que possuam maior chance de revelar defeitos do tipo usos ausentes, ou seja, um uso que deveria existir e foi omitido pelo programador.

Esses critérios apresentam diversas propriedades interessantes, dentre elas: estabelecem uma hierarquia de critérios entre os critérios todos os ramos e todos os caminhos, mesmo na presença de caminhos não executáveis; nenhum outro critério baseado em análise de fluxo de dados *inclui* [10] os critérios Potenciais Usos; e, requerem na prática um número relativamente pequeno de casos de teste [Mal91].

Não serão mostradas as definições desses critérios. Tais definições, assim como uma conceituação mais completa relativa à análise de fluxo de dados e critérios de teste estrutural podem ser obtidas em [Mal91].

---

[9] No critério Análise de Mutantes a idéia de *score de mutação* é análoga ao *nível de cobertura* para critérios estruturais. Esse valor refere-se à porcentagem de programas mutantes distinguidos do original — ou "mortos".

[10] Informalmente um critério $c1$ *inclui* um critério $c2$ se qualquer conjunto de caminhos completos T que satisfaz $c1$ também satisfaz $c2$.

2.7 Sumário

Foram apresentados neste capítulo conceitos essenciais de Teste de Software e detalhadas as definições básicas de teste estrutural. Estes são conceitos utilizados genericamente neste trabalho, mas não são suficientes para o entendimento do mesmo. Conceitos adicionais (não necessariamente de teste) serão apresentados ao longo da dissertação quando forem necessários.

Pode-se relacionar o tema deste trabalho à atividade de *projeto de casos de teste* descrita anteriormente. Esta atividade envolve a adoção de um critério de teste; a aplicação do critério ao programa, gerando assim os elementos requeridos no teste; e a seleção de dados de teste para o programa tais que o critério selecionado seja satisfeito.

Conforme destacado por Maldonado [Mal91], a aplicação de critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muitos simples.

A aplicação do critério ao programa é uma tarefa freqüentemente automatizada, apoiada por ferramentas como a ATAC [HL90] e a POKE-TOOL [Cha91]. Essencialmente é feita uma análise estática do código fonte, através da qual são identificados os elementos requeridos. Adicionalmente são produzidas informações para a avaliação posterior do nível de cobertura do teste e gerado o programa instrumentado para o teste [11].

Tanto a geração dos elementos requeridos quanto a análise de cobertura são tarefas automatizáveis. Entretanto, a automação completa da geração de dados de teste é inviável; isto requereria a solução de problemas clássicos indecidíveis como o "problema da parada" [Cla76, How75].

Por outro lado, esta automação é altamente desejável, visto que o processo de gerar manualmente dados de teste é difícil, tedioso e sujeito a erros. Neste sentido diversos autores realizaram pesquisas e propuseram abordagens para que seja conseguida alguma automação no processo de geração de dados de teste.

No capítulo seguinte são discutidos trabalhos abordando a automação da geração de dados para o teste estrutural de software.

[11] O programa instrumentado contém comandos que permitem o monitoramento do teste; este aspecto é melhor descrito posteriormente.

Capítulo 3

Geração Automática de Dados de Teste para a Satisfação de Critérios de Teste Estrutural

No teste estrutural satisfazer um critério significa exercitar todos os componentes estruturais requeridos pela aplicação do critério ao programa em teste. Para que isto ocorra é necessário que, para cada elemento requerido, algum dado do conjunto de dados de teste o exercite.

Portanto, neste contexto, a geração de dados de teste visa a selecionar pontos do domínio de entrada do programa para satisfazer um dado critério. A tarefa de selecionar esses dados requer do testador conhecimentos específicos do critério utilizado. Além disso, requer uma cuidadosa análise do código do programa em teste. Isto faz com que a geração de dados de teste seja uma atividade de difícil realização e tenda a ter um alto impacto no custo final do teste, sobretudo quando da utilização de critérios exigentes (que requerem quantidade de dados substancial para serem satisfeitos), como os critérios Potenciais Usos [MCVJ91].

A geração de dados de teste é dependente do critério de teste utilizado [1]. Para cada critério é necessário identificar as características dos elementos requeridos pelo critério e os requisitos necessários para que estes elementos sejam exercitados. Exemplificando: Para que um mutante seja morto no critério Análise de Mutantes [2] [DMLS78] é necessário que sejam satisfeitas as condições de alcançabilidade, necessidade e suficiência [OP95]. Os critérios baseados em análise de fluxo de dados definidos em [RW85] requerem basicamen-

[1] Deve-se salientar que a geração aleatória de dados é independente do critério de teste utilizado, sendo aplicável a diversos critérios.

[2] Apesar do critério Análise de Mutantes ser baseado em defeitos, a automação da geração de dados para este critério passa pelo tratamento de problemas análogos aos existentes no caso dos critérios de teste estrutural [Off96].

te que dados de teste executem sub-caminhos livres de definição de cada nó contendo uma definição de uma variável, para nós contendo usos computacionais, e arcos contendo usos predicativos desta variável. Os critérios de teste Potenciais Usos [Mal91] requerem basicamente que, para toda variável definida no programa, sejam executados sub-caminhos livres de definição com respeito a esta variável do nó de definição para todo nó e para todo arco possível de ser alcançado a partir do nó de definição. Este tipo de análise pode ser estendida aos critérios definidos por Ntafos [Nta84] e Laski & Korel [LK83].

Esforços têm sido conduzidos por diversos autores no sentido de automatizar a geração de dados de teste [BEK75, MJM75, Cla76, How77, RSBC76, MS76, BM83, Kor90, WGS94, Kor96, OP95, Off96, FK96, PH97, GN97, GBR98, GMS98]. A automação da atividade de geração de dados no teste estrutural é uma tarefa difícil e que envolve uma série de problemas complexos. Na verdade, o problema, em uma abordagem geral, não tem solução. O problema de gerar dados para executar um determinado comando em um programa é equivalente ao problema da parada ("halting problem") e, portanto, indecidível [Cla76, How75].

Outra questão complexa, associada ao problema da parada, diz respeito à existência de caminhos não executáveis: caminhos para os quais não existe algum conjunto de valores de entrada do programa que os executem. A existência desses caminhos pode fazer com que elementos requeridos pelos critérios sejam não exercitáveis. Nesses casos, para a satisfação dos critérios, é necessária a identificação desses elementos. Determinar automaticamente caminhos não executáveis — ou mais genericamente elementos requeridos não executáveis — também é uma questão indecidível no geral, apesar de existirem soluções parciais. Este problema vem sendo tratado por vários autores [Fra87, MYV90, HH85, Ver92, BJVM97, FB97, BGS97] e é abordado na Seção 4.7.

É possível, quando da aplicação de critérios de teste estrutural (inclusive os baseados em análise de fluxo de dados), caracterizar um conjunto de caminhos completos no programa que, quando executados, fazem com que os elementos requeridos pelos critérios sejam exercitados. Uma estratégia válida, portanto, é dividir o processo de geração de dados de teste em duas etapas:

[i] Seleção de um caminho completo no programa que exercite cada elemento requerido pelo critério de teste; e

[ii] Geração de um dado de entrada que execute cada caminho selecionado.

Nesta estratégia tem-se a etapa [i] dependente do critério, enquanto a [ii] é genérica. Isto significa que um gerador automático de dados de teste que executem caminhos completos do programa pode ser utilizado para diversos critérios de teste estrutural. Isto inclui os baseados em análise de fluxo de dados, descritos acima e critérios baseados em análise de fluxo de controle, cujos elementos requeridos são arcos, nós ou caminhos no grafo do programa.

A seleção de caminhos para satisfação de critérios de teste estrutural (etapa [i]) vem sendo abordada por vários autores [MYV90, Ver92, FB97, BGS97, Per97]. Este problema não será tratado diretamente neste trabalho; entretanto, por estar relacionado à questão da não executabilidade de caminhos, será contemplado na Seção 4.7.1.

Especificamente quanto à geração de dados para a execução de caminhos no programa (etapa [ii] mencionada ateriormente) duas abordagens podem ser identificadas na literatura: a execução simbólica [Cla76, How77] e a técnica dinâmica de geração de dados de teste [Kor90, FK96, GN97].

Ambas as abordagens buscam encontrar algum ponto do domínio de entrada que satisfaça a expressão simbólica associada ao caminho que pretende-se executar. Esta expressão, referida como *expressão de caminho* [Off96] ou *condição de caminho* [GN97], reflete os requisitos necessários para que um conjunto de dados de entrada provoque a execução do caminho. A expressão de caminho consiste na conjunção dos predicados encontrados ao longo do mesmo e envolve variáveis do programa, constantes, operadores relacionais e lógicos. As variáveis de entrada do programa influenciam os predicados direta ou indiretamente através de "cadeias de interações de fluxo de dados" [Nta84]. Deste modo, as expressões de caminho podem ser vistas como expressões simbólicas envolvendo as variáveis de entrada do programa.

Os trabalhos que utilizam execução simbólica buscam representar simbolicamente as condições para a execução de um dado caminho em função das variáveis de entrada do programa. Esta representação simbólica é utilizada por algoritmos que tentam buscar soluções as quais, satisfazendo as condições, provoquem a execução do caminho pretendido.

A técnica dinâmica é baseada na execução real do programa, em métodos de minimização de funções e análise dinâmica de fluxo de dados.

Trabalhos seguindo estas duas linhas são descritos a seguir.

3.1 Geração de Dados de Teste Através de Execução Simbólica

Execução simbólica e sua aplicação ao teste de programas não são idéias novas. Durante as duas últimas décadas, vários pesquisadores têm se dedicado a esse tema [Cla76, How77, BEK75, RSBC76, PH97]. Geralmente, ela é utilizada conjuntamente com a técnica baseada em caminhos e tem o objetivo de auxiliar a geração automática de dados de teste para um dado caminho ou conjunto de caminhos. Pode, em alguns casos, detectar caminhos não executáveis e, além disso, criar representações simbólicas para as variáveis de saída na execução de um caminho que poderão ser comparadas com representações simbólicas das saídas esperadas, facilitando a detecção de um defeito. A execução simbólica tem sido

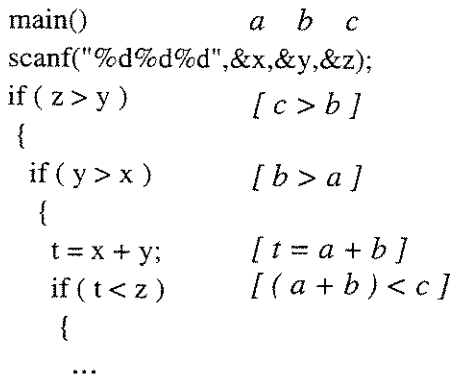

Figura 3.1: Geração de Dados de Teste Através de Execução Simbólica

utilizada também em outras áreas da computação, tais como a depuração de programas [CR83].

Técnicas de execução simbólica derivam expressões algébricas que representam a execução de um dado caminho (ou conjunto de caminhos). O resultado é uma expressão simbólica chamada *computação do caminho* que representa as variáveis de saída em termos das variáveis de entrada [Fra87], e uma expressão, chamada *condição do caminho*, que representa condições para que o caminho seja executado.

A condição do caminho é dada pelo conjunto de restrições associadas a todos os predicados encontrados ao longo do mesmo. O *domínio do caminho* é dado pelo conjunto de valores que satisfazem a condição do caminho. O dado de teste é então gerado escolhendo-se um elemento desse conjunto. Nem sempre é possível determinar se existe uma solução que satisfaça a condição do caminho, ou seja, determinar se o caminho é ou não executável. Também não é garantido que para todos os casos nos quais existe alguma solução ela será descoberta.

A Figura 3.1 ilustra a idéia geral desta técnica. Nesta figura os valores simbólicos a, b e c são atribuidos às variáveis x, y e z, respectivamente. Estes valores são processados simbolicamente de modo que as variáveis internas e os predicados do programa sejam representados em função das variáveis de entrada (no exemplo a variável t é equivalente a $a + b$ e o primeiro predicado equivalente a $c > b$). A conjunção dos predicados do caminho que deseja-se executar representa a condição que precisa ser satisfeita para que o caminho seja executado. No exemplo $(c > b)$ *and* $(b > a)$ *and* $((a + b) < c)$ determina a condição suficiente para que os três predicados existentes sejam satisfeitos.

Encontra-se em [BJ97] um estudo detalhado das diversas propostas já feitas nesta abordagem. Podem ser destacados os seguintes trabalhos:

- Clarke [Cla76] propõe uma feramenta com as seguintes funcionalidades principais:

Gerar dados de teste para provocar a execução de um caminho no programa; detectar alguns caminhos não executáveis e criar uma representação simbólica das variáveis de saída do programa como função das variáveis de entrada. São gerados dados de teste apenas quando as restrições associadas ao caminho são lineares. Isto porque para satisfazer, de uma forma geral, os comandos condicionais em um caminho, é necessário que o sistema seja capaz de resolver sistemas de inequações arbitrários, o que, segundo a autora, é um problema indecidível. A geração de dados não é possível também na presença de estruturas do tipo vetor.

- Ramamoorthy e outros [RSBC76] propõem uma abordagem bastante semelhante à anterior, implementada no protótipo "CASEGEN". É feito o processamento simbólico do programa, resultando em um conjunto de equações e inequações em função das variáveis de entrada. Para a geração de dados de entrada é utilizada a "tentativa e erro sistemática". A abordagem segue uma série de passos: Inicialmente as restrições estruturais são descritas em forma conjuntiva. As variáveis de entrada são organizadas em uma seqüência específica. Cada variável de entrada é associada de forma sistemática às restrições que as incluem. Baseado nessas associações são atribuídos valores aleatórios de entrada, considerando uma variável por vez. Sempre que não é satisfeita alguma cláusula é feito um "backtracking" e são atribuidos novos valores. Tem-se portanto um processo iterativo que termina quando todas as variáveis forem definidas para satisfazer as cláusulas ou, caso estes valores não sejam encontrados, tem-se a falha na geração de dados.

- Howden [How77, How78] propõe o Sistema DISSECT que suporta a avaliação simbólica de programas em Fortran. Dois arquivos são entradas do sistema: um contendo o programa a ser testado e outro uma lista de "comandos" que controlam a avaliação simbólica, permitindo a realização do teste. Os comandos de entrada permitem que valores reais ou simbólicos sejam atribuídos às variáveis durante a execução do programa. Comandos de saída permitem que os valores das variáveis ou dos predicados associados aos caminhos do programa sejam fornecidos ao testador. É introduzida uma abordagem para o tratamento de vetores baseada em lista de valores simbólicos atribuídos e tratamento de referências ambíguas. É feita também uma análise da adequação do teste simbólico às classes de defeitos. Não é abordada a geração automática de dados que executem os caminhos processados simbolicamente.

- Boyer e outros apresentam um sistema experimental — o SELECT, compatível com um subconjunto da linguagem LISP, que faz o processamento dos caminhos do programa com os seguintes objetivos: gerar dados de teste; prover valores simbólicos para as variáveis do programa como resultado da execução dos caminhos; e, provar a corretude de um caminho com respeito às asserções informadas pelo usuário [BEK75]. A geração dos dados de teste é feita por um algoritmo de gradiente conjugado. Este algoritmo busca minimizar uma função potencial construída a partir

das inequações simbólicas geradas. Um aspecto negativo da abordagem é que ela requer interação com o usuário, além de não terminar garantidamente.

- Price e outros [PH97] propõem a ferramenta LOGTEST, que apóia o teste estrutural de programas em Pascal e possui como recursos principais: análise do programa e construção de uma base de conhecimento; seleção de caminhos para teste; geração de "slices" estáticos e dinâmicos e depuração baseada em conhecimento. A partir da base de conhecimento inicial sobre o programa — que contém informações sobre o fluxo de dados e de controle, são geradas as cláusulas para execução simbólica. Como resultado da execução simbólica têm-se os predicados de caminho e os contextos das variáveis. A representação é feita em função das variáveis de entrada. Tem-se para a geração de dados um processo iterativo envolvendo: seleção de um conjunto de sub-caminhos de acordo com um critério; execução do programa com dados reais; análise de cobertura e seleção de um sub-caminho para o teste; execução simbólica do caminho executado anteriormente com dados reais; análise e substituição de valores simbólicos por valores reais a fim de forçar a execução do sub-caminho selecionado. A abordagem apresentada distingüe-se das demais por combinar execução simbólica e real. Não são apresentados resultados de validação da ferramenta.

3.1.1 Aspectos de Difícil Tratamento

A execução simbólica apresenta alguns aspectos de difícil tratamento que restringem a sua aplicabilidade:

O tratamento de laços tende a ser complexo. Quando o número de iterações no laço não é determinado é necessário deduzir os valores das variáveis no laço, o que pode gerar relações de recorrência de difícil solução. Representar de forma completa a "condição de alcançabilidade" de um nó é indecidível devido à possibilidade de existirem muitos caminhos que levam ao nó, gerados pela existência de um laço no programa [OP95];

Existem problemas quando há referências às variáveis compostas (vetores e matrizes) e aos ponteiros. O problema é identificar a qual variável se faz referência. As soluções existentes são variações para o tratamento das ambiguidades geradas, sendo sempre necessária a intervenção do testador para resolvê-las;

O tratamento de chamadas de outros módulos por processamento simbólico tende a gerar equações muito extensas e complexas. Representar em uma expressão o resultado da execução de um módulo equivale a realizar avaliação simbólica global, que possui grandes restrições de implementação [Fra87]. Soluções propostas em [How77] podem levar a uma explosão do número de caminhos e condições geradas, ou a geração de condições inconsistentes.

Buscando um melhor tratamento desses aspectos foi proposta a técnica dinâmica de geração de dados de teste, abordada a seguir.

3.2 Técnica Dinâmica de Geração de Dados de Teste

A técnica dinâmica de geração de dados de teste é baseada na execução real do programa, em métodos de minimização de funções e análise de fluxo de dados dinâmica.

A Figura 3.2 ilustra esta técnica. No processo de geração de dados de teste valores gerados aleatoriamente são atribuidos às variáveis de entrada e o programa é executado. Caso o fluxo de execução desvie do pretendido em algum predicado, técnicas de otimização são aplicadas para modificar os valores de entrada visando provocar a execução do fluxo de controle desejado. Exemplificando: caso seja desejada a satisfação do predicado $(z > y)$ e isto não ocorra na execução, é associada ao predicado uma função $E1 = y - z$. Esta função permite apurar o "erro" que fez com que o predicado não fosse tomado como o desejado. O valor $E1$, obtido através do monitoramento da execução, pode ser utilizado para direcionar as mudanças nas variáveis de entrada visando a satisfação do predicado em questão; mudanças nessas variáveis que minimizam o valor de $E1$ estão na "direção correta", isto é, podem levar à satisfação do predicado.

No programa da Figura 3.2 caso as variáveis de entrada tenham os valores $x = 2$, $y = 5$ e $z = 3$ o predicado $(z > y)$ não será satisfeito e será computado o valor $E1 = y - z = 2$. Para os valores $x = 2$, $y = 5$ e $z = 4$ o valor computado para $E1$ será menor $(E1 = y - z = 1)$, o que significa que o predicado está mais "próximo de ser satisfeito". Para os valores $x = 2$, $y = 5$ e $z = 6$ o predicado $(z > y)$ será satisfeito, assim como o predicado $(y > x)$; o predicado $(t < z)$, por outro lado, não será satisfeito. Será necessário portanto realizar uma nova busca por valores para x, y e z que minimizem $E3$ a fim de satisfazer o predicado $(t < z)$. Deve-se notar que, na tentativa de satisfazer o terceiro predicado, existe a restrição de que os dois predicados anteriores devem continuar sendo satisfeitos; ou seja, as mudanças nas variáveis de entrada não devem mudar as avaliações dos predicados anteriores.

3.2.1 Miller e outros

A técnica dinâmica foi proposta inicialmente por Miller e outros [MS76]. Os autores propõem a solução do problema de geração de dados de teste como um problema de otimização numérica. Os comandos de decisão do programa são trocados por comandos chamados "restrições de caminho" que medem o quão próxima cada decisão está de ser satisfeita. O programa toma a forma de uma única linha de código com uma série de restrições cujos erros devem ser minimizados. Variáveis de entrada cujo tipo não seja

```
main()
scanf("%d%d%d",&x,&y,&z);
if ( z > y )                [ E1 = y - z ]
 {
  if ( y > x )              [ E2 = x - y ]
   {
    t = x + y;
    if ( t < z )            [ E3 = t - z ]
     {
      ...
```

Figura 3.2: Técnica Dinâmica de Geração de Dados de Teste

real devem ser atribuidas pelo testador. A busca por dados de entrada que executem o caminho, para as variáveis do tipo real, é realizada por algoritmos de busca direta.

3.2.2 Korel e outros

Korel [Kor90] desenvolveu um protótipo denominado TESTGEN, compatível com um sub-conjunto do Pascal, que utiliza a técnica no intuito de encontrar valores para as variáveis de entrada do programa tais que um determinado caminho seja executado. Dados reais são atribuídos às variáveis de entrada e o fluxo de execução do programa é monitorado. Se um ramo incorreto foi tomado, métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais o ramo correto seria tomado. A abordagem considera predicados simples e lineares da forma E_1 *op* E_2. Todo predicado pode ser transformado na forma F *rel* 0, onde $rel \in \{<, \leq, =\}$. Esta abordagem permite o tratamento de estruturas do tipo vetor.

O problema inicial é determinar um conjunto x^0 de valores de entrada que provoquem a execução de um dado caminho $P = (n_1, .., n_i, ..., n_m)$. Korel diz que esse problema pode ser reduzido a uma seqüência de "sub-objetivos", onde cada sub-objetivo será resolvido utilizando-se técnicas de minimização de funções. Se P é executado com o conjunto x^0 de valores de entrada, então x^0 é a solução para o problema de geração. Senão foi executado um outro caminho T e x^0 não é a solução. Neste caso $P_1 = (n_1{}^p, n_2{}^p, ...n_i{}^p)$ é o mais longo sub-caminho de T, tal que $(n_1{}^p = n_1 ... n_i{}^p = n_i)$, e o caminho correto não foi executado porque o predicado associado ao arco (n_i, n_{i+1}) não foi avaliado como se esperava. Então o objetivo é encontrar valores de entrada x^1 tais que $F_i(x)$ rel_i 0 seja satisfeita, provocando a execução do ramo (n_i, n_{i+1}) como pretendido. A função $F_i(x)$ pode ser minimizada até que ela se torne negativa (ou 0 dependendo de rel_i). Uma vez resolvido o primeiro sub-objetivo, outros deverão ser resolvidos até que a solução do objetivo principal seja

encontrada, ou ainda, até que se decida que o sub-objetivo não poderá ser resolvido (nesse caso a pesquisa falha).

A busca dos valores é feita pelo "método da variável alternativa", que consiste em minimizar com respeito a somente uma variável de entrada por vez. Se, considerando uma certa variável x_i, a função $F_i(x)$ não se torna negativa, as demais variáveis de entrada são consideradas, uma por vez. Neste processo são feitos "movimentos exploratórios" a fim de definir como alterar a variável considerada para que $F_i(x)$ diminua. Se esses movimentos indicam uma direção a ser tomada, uma mudança de maiores proporções (aumento ou decremento da variável x_i), é executada. A pesquisa falhará se todas as variáveis forem analisadas e a função $F_i(x)$ não puder ser decrementada. É mostrada também uma adaptação dessas idéias para programas que manipulam estruturas dinâmicas de fluxo de dados.

Esta técnica foi estendida por Ferguson de forma a incorporar a seleção de caminhos no processo de geração de dados [FK96]. São utilizadas informações sobre dependência de dados para determinar comandos do programa que afetam a execução de outros. Deste modo, são definidas "seqüências de eventos" (seqüências de nós) que devem ser executadas antes de atingir um dado predicado, de forma que o algoritmo de busca tenha maior chance de satisfazê-lo.

Não é tratada a questão da não executabilidade de caminhos. Segundo o autor "a habilidade muito limitada da abordagem dinâmica para tratar a questão da não executabilidade é um dos maiores problemas deste tipo de abordagem" [Kor90]. Conforme descrito no Capítulo 4, é possível definir um tratamento alternativo que explore potencialidades inerentes à técnica dinâmica em relação ao problema da não executabilidade de caminhos. Outros aspectos não tratados são os predicados compostos com operadores lógicos; estruturas de seleção múltipla e variáveis dos tipos caráter e "string".

3.2.3 Gallagher e outros

Gallagher e outros [GN97] apresentam o ADTEST, um sistema para geração automática de dados de teste para programas escritos em Ada. A abordagem utilizada baseia-se na execução do programa e na aplicação de otimização numérica, assim como as descritas anteriormente. É feita a transformação do problema de geração de dados no problema de "otimização numérica não linear com restrições". É empregada a técnica de "funções de penalidade" para reduzir este problema ao de "otimização sem restrições". É criada uma função $F(x,w) = \sum_{i=1}^{n} G(g_i(x), w_i, type_i)$ onde w_i ... w_n são pesos positivos e o termo $G(g_i(x), w_i, type_i)$ representa a restrição imposta pelo predicado i do caminho. Valores $g_i(x)$ são baseados nos operadores relacionais $type_i$ envolvidos e são determinados dinâmicamente pela execução do programa instrumentado. A função G assume valores pequenos se o predicado é satisfeito e valores maiores, caso contrário. Uma técnica de

otimização "quasi-Newton" é empregada para minimizar $F(x, w)$. Durante a busca, à medida que os predicados são satisfeitos, novos predicados do caminho e respectivos valores $G(g_i(x), w_i, type_i)$ são considerados na otimização.

Aspectos positivos incluem: o tratamento de predicados compostos envolvendo operadores lógicos; a adoção de uma técnica de otimização mais eficiente (segundo os autores) em relação à utilizada em [Kor90]; o tratamento de estruturas não numéricas como caracteres (além de valores inteiros e reais). Entretanto, o tratamento de "strings" é feito representado-os pela soma dos caracteres e não como elementos distintos. Isto é certamente problemático pois dois strings distintos podem ter soma semelhante, sendo considerados erroneamente como idênticos. Também não é explicitado o tratamento de variáveis do tipo vetor.

3.2.4 Gupta e outros

Gupta e outros [GMS98] utilizam uma abordagem baseada em execução denominada "Método de Relaxamento Iterativo". O objetivo é gerar dados de teste que provoquem a execução de um caminho pré-definido. O método inicia com a aplicação de valores de entrada gerados aleatoriamente; se o caminho desejado não é executado então a entrada é refinada de forma iterativa.

Para aplicação do método são derivadas duas representações para cada predicado do caminho: o *"slice" do predicado* é o sub-conjunto de entradas e comandos de atribuição que precisam ser executados para a avaliação do predicado. Esta é uma representação exata da função computada pelo predicado. Usando esta representação é derivada a *representação linear aritmética* do predicado em termos das variáveis de entrada do programa. Esta representação é gerada para permitir a aplicação de técnicas de análise numérica. Se a função computada pelo predicado é linear com respeito às variáveis de entrada, então a representação linear aritmética é exata; senão, esta representação aproxima o valor da função associada ao predicado na sua vizinhança.

Estas duas representações são utilizadas para refinar os valores de entrada iniciais a fim de obter a entrada desejada do seguinte modo: Caso a execução do "slice" do predicado determine que o mesmo não foi avaliado como desejado, a avaliação da função computada pelo predicado provê um valor chamado *resíduo do predicado*. Este resíduo é a quantidade que o valor da função precisa mudar para que o predicado seja avaliado como desejado. Utilizando este valor e a representação linear aritmética do predicado é derivada a *restrição linear nos incrementos* para entrada atual. Esta restrição é derivada para cada função de predicado do caminho. Todas as restrições são então resolvidas simultaneamente usando "Gaussian Elimination" (um método para solução de sistemas lineares) para computar os incrementos na entrada atual.

Neste método, caso as funções de predicado sejam todas lineares em função das va-

riáveis de entrada, é gerada a entrada desejada em uma iteração ou é garantido que o caminho é não executável. Entretanto, basta que alguma função de predicado seja não linear para que a geração falhe nesta tentativa. Nestes casos podem ser usados diversos refinamentos sucessivos para que a avaliação do predicado seja a desejada. Em cada refinamento tem-se a execução dos "slices" computados para todos os predicados do caminho e a computação dos valores: representação linear aritmética e resíduo, para os todos predicados.

Um aspecto positivo desta abordagem é que, para uma classe de problemas, (caminhos com funções de predicado lineares) tem-se um tratamento bastante eficiente, tanto para a geração de dados de entrada quanto para a identificação da não executabilidade do caminho. Outro ponto interessante é que o conceito de slice do predicado permite a avaliação de todos os predicados do caminho simultaneamente, mesmo dos que não seriam alcançados com os valores de entrada aplicados. Isto tende a eliminar a necessidade de operações "backtracking" existente quando considera-se cada predicado por vez.

Aspectos negativos são: é afirmado ser possível estender a técnica para entradas não numéricas, mas não é definido como fazê-lo. A cada "tentativa" para geração de dados é necessário executar os diversos "slices" computados e realizar um considerável processamento numérico para obtenção dos novos valores de entrada. Trata-se de um método descrito através da análise de um programa específico; não é mencionada a implementação de um protótipo, nem são fornecidos dados de validação. É afirmada a necessidade de introduzir predicados para evitar erros de execução (verificação de valores limites para vetores, divisões por zero, etc.), entretanto esses predicados não são considerados no exemplo, nem é descrito como automatizar este tratamento. A abordagem de forçar a execução de comandos presentes nos "slices", ignorando os predicados anteriores aos comandos, pode ser um sério problema na implementação do método proposto. Por exemplo: os "slices" computados podem conter comandos do tipo: $y = \frac{f(x)}{g(x)}$ mas excluirem um predicado anterior do tipo $if(g(x)! = 0)$, cuja função seria a de identificar esta condição e permitir um tratamento correto. Neste caso, executar o slice com valores de entrada tais que $g(x) = 0$, provocará um erro de execução e impossibilitará a avaliação do predicado.

3.2.5 Aspectos Comuns

Nas abordagens elaboradas por estes autores, ao invés de representar simbolicamente os predicados do programa a fim de satisfazê-los (para executar o caminho pretendido), é feita uma avaliação real do valor do predicado usando para tanto a execução do programa. A partir desta avaliação real são usados métodos de minimização, buscando os valores que executem o caminho pretendido. Essencialmente, como destacado em [MS76], o próprio programa fornece um meio eficiente de avaliar numericamente o erro nos predicados que fazem com que o caminho executado desvie do pretendido.

Desta forma, na abordagem dinâmica a interferência de chamadas de outros módulos e variáveis compostas nos predicados (aspectos críticos para o processamento simbólico) é apurada a partir de valores reais das variáveis na execução e não através de expressões simbólicas. Isto faz com que tais problemas sejam contornados, deixando a tarefa de busca dos dados para executar o caminho para o algoritmo de busca utilizado.

O tratamento de laços é feito de forma transparente, no sentido de que não há necessidade de distinguir se um ramo está inserido em algum laço ou não. Em qualquer caso, se o ramo incorreto foi tomado em relação ao pretendido, métodos de minimização de funções são utilizados para determinar valores para as variáveis de entrada para os quais o ramo correto é tomado.

3.2.6 Pontos Críticos e Motivação do Trabalho

Deve-se ter em mente que a solução completa do problema de geração de dados no teste estrutural é inviável, por motivos já relatados. Deste modo, qualquer abordagem é, de certa forma, uma tentativa de obtenção de soluções parciais, aplicáveis na maioria das situações reais e úteis do ponto de vista prático.

Sob esta ótica são levantados pontos que podem ser melhor estudados:

[i] Um aspecto crítico relaciona-se ao tratamento da questão da não executabilidade: inviável segundo Korel [Kor90], pouco enfatizado por Gallagher e outros [GN97] e tratado de forma restrita aos caminhos com predicados lineares por Gupta e outros [GMS98];

[ii] Em todas as abordagens tem-se um procedimento iterativo, no qual é possível que sejam geradas diversas soluções incorretas até que os valores de entrada desejados sejam obtidos. Estas soluções incorretas contribuem para que os valores desejados sejam encontrados. Em nenhum dos trabalhos, entretanto, tais soluções são mantidas como um "conhecimento" agregado sobre o programa, potencialmente útil em etapas posteriores na geração de dados (leia-se: nos próximos caminhos a serem testados);

[iii] O ponto inicial para a busca é gerado aleatoriamente em todos os casos. Embora destacado que a utilização pontos iniciais "distantes" da solução podem provocar a falha da busca [GN97] não são apresentadas soluções para que este risco seja minimizado;

[iv] Predicados do programa podem representar funções complexas das variáveis de entrada, que necessitam ser resolvidas para que o caminho desejado seja executado. Como, em última análise, tais funções relacionam-se à semântica dos programas,

não é possível assumir que elas possuam características como: linearidade, continuidade, existência de derivadas, ou que sejam unimodais. Isto porque problemas reais não possuem obrigatoriamente tais características. São requeridos, portanto, métodos de busca genéricos e robustos, que funcionem de forma eficaz não apenas para classes específicas de problemas.

3.3 Contexto da Ferramenta de Geração Automática de Dados de Teste

Na seção anterior foram destacados diversos trabalhos relacionados à geração automática de dados de teste. Nesta seção é descrito o contexto no qual a ferramenta proposta neste trabalho está inserida.

3.3.1 Ferramenta POKE-TOOL

A ferramenta de geração de dados de teste está inserida no contexto da ferramenta de teste POKE-TOOL [Mal91, Cha91, BCM+95], que apóia a aplicação dos critérios de teste Potenciais Usos [Mal91] para o teste de programas escritos em linguagem C na versão atualmente em uso.

A Ferramenta POKE-TOOL constitui-se de diversos módulos integrados através de "scripts shell" e com o acionamento realizado por linha de comando, tendo também uma versão com interface gráfica. A versão mais atual da ferramenta é compatível com o sistema operacional UNIX.

O comando *poketool* aciona a análise estática do código fonte. Como resultado da execução tem-se: a instrumentação do programa para o teste; a geração dos elementos requeridos para os critérios apoiados e de informações estáticas do programa, além do grafo de fluxo de controle do programa em teste.

O comando *pokeexec* permite a execução do programa instrumentado, direcionando para arquivos: os dados de entrada; os parâmetros fornecidos; as saídas obtidas; e os caminhos executados para cada caso de teste.

O comando *pokeaval* permite a avaliação da cobertura atingida com o conjunto de casos de teste aplicados, para o critério escolhido, além de fornecer os elementos requeridos executados e os não executados.

Maldonado destaca a arquitetura [Mal91] e Chaim [Cha91] detalha os aspectos funcionais e de implementação desta ferramenta.

3.3.2 Pokeheuris, Pokepadrão e Pokepaths

Estão disponíveis módulos de apoio ao tratamento de elementos não executáveis. De um modo geral esses módulos possibilitam a aplicação de heurísticas e execução simbólica [Ver92, VMJ92], através do módulo *pokeheuris*.

É apoiado também o tratamento de padrões de não executabilidade[3] — módulo *pokepadrão* — [BJVM97], que retira do conjunto de elementos requeridos aqueles que são não executáveis devido a algum padrão identificado pelo testador.

Estes módulos são importantes porque reduzem o trabalho do testador na identificação de elementos não executáveis, fator crítico em termos de custo [VMJ92].

Está sendo implementado também um módulo que gera caminhos completos para cobrir os elementos requeridos pelos critérios Potenciais Usos, o *pokepaths*. Perez [Per97] analisa aspectos relacionados à seleção de caminhos para a cobertura de critérios estruturais de teste.

3.3.3 Utilização da Ferramenta neste Contexto

Uma abordagem para a utilização da ferramenta de geração automática de dados de teste neste contexto é:

i. Utilizar o comando *poketool* para aplicar o critério de teste selecionado ao programa, gerando um conjunto de elementos requeridos para o teste, segundo o critério escolhido;

ii. Utilizar o comando *pokeheuris* para identificar alguns elementos requeridos não executáveis e o *pokepadrão* para remover do conjunto de elementos requeridos os que contenham algum padrão de não executabilidade identificado;

iii. Utilizar o comando *pokepaths* para gerar caminhos completos que cubram cada um dos elementos requeridos;

iv. Utilizar a ferramenta de geração automática de dados de teste para obter os dados de entrada que executem cada um dos caminhos completos selecionados;

De qualquer modo a utilização da ferramenta deve ser feita tendo em mente o *plano de teste* estabelecido. É possível, por exemplo, selecionar dados de entrada através de algum critério de teste funcional, ou ainda, gerar dados de entrada aleatoriamente; avaliar a

[3]Informalmente *padrões de não executabilidade* são componentes estruturais identificados como sendo não executáveis. Tais padrões podem determinar a não executabilidade de diversos elementos requeridos pelos critérios.

cobertura atingida utilizando o *pokeaval*; e só então, para os elementos requeridos ainda não executados, utilizar o procedimento descrito acima. Em suma, a utilização poderá se dar tanto para gerar dados buscando a satisfação do critério de teste selecionado, quanto para complementar o nível de cobertura atingido no teste com uma massa de dados pré-existente.

Não é objetivo deste trabalho estudar *planos* ou *estratégias* de teste; portanto apenas identificou-se como inserir a ferramenta de geração automática de dados de teste no ambiente POKE-TOOL. Vergilio [Ver97] aborda esses temas e define uma estratégia incremental para a geração de dados de teste.

3.4 Sumário

Foram descritas neste capítulo as principais linhas de pesquisa relacionadas à automação da geração de dados no teste estrutural de programas.

Foram destacados trabalhos que utilizam processamento simbólico do programa e as mais rescentes abordagens, que utilizam a técnica dinâmica. Devido à maior generalidade desta última a técnica dinâmica foi escolhida para o modelo proposto neste trabalho.

Foram identificados pontos críticos não contemplados pelas abordagens existentes para geração automática de dados de teste presentes na literatura. Tais pontos determinaram reflexões a partir das quais foi proposto o modelo da feramenta de geração automática de dados de teste proposta neste trabalho.

No capítulo seguinte tem-se a descrição da abordagem proposta nesta dissertação.

Capítulo 4

Uma Ferramenta de Geração Automática de Dados de Teste e Tratamento de Não Executabilidade

Este capítulo descreve as soluções desenvolvidas para o tema proposto nesta dissertação. São destacados os aspectos conceituais relacionados à abordagem proposta e descritos detalhes do modelo da ferramenta desenvolvida.

4.1 Visão Inicial

A ferramenta é um gerador de dados de entrada para executar caminhos completos no programa em teste. É utilizada a execução real do programa (técnica dinâmica [Kor90]) para determinar se o caminho pretendido foi executado ou não e para direcionar a busca da solução. É utilizado um algoritmo de busca para avaliar os caminhos realmente executados e realizar mudanças nas variáveis de entrada no objetivo de encontrar os dados que executem o caminho pretendido. É mantida uma "base de soluções" consultada sempre que o testador define um novo caminho a ser executado. Este arquivo contém informações das execuções anteriores de caminhos do programa e pode possibilitar a redução do custo da busca. Descreve-se sucintamente a seguir o funcionamento da ferramenta.

Inicialmente tem-se a instrumentação do programa em teste. São inseridas "pontas de prova" com o objetivo de monitorar a execução do programa para que sejam produzidas as informações: seqüência de nós executada no programa (caminho executado); e os valores das variáveis utilizadas na avaliação dos predicados percorridos na execução do programa.

O testador fornece à ferramenta os parâmetros de controle da busca; dados sobre a manutenção da base de soluções e sobre a interface do programa em teste (número de

variáveis e/ou parâmetros de entrada e respectivos tipos e faixas de valores). Devem ser informados também: nome do arquivo do programa em teste executável (versão instrumentada e compilada), além do nome do arquivo onde estão os *caminhos pretendidos*, isto é, caminhos para os quais se deseja descobrir dados de entrada os executem.

Os caminhos pretendidos são tratados um por vez. Considerando o caminho selecionado, é feita uma busca na base de soluções para verificar se este caminho já foi executado anteriormente; caso positivo, é recuperado o conjunto de valores de entrada que provocou a sua execução, que é a solução desejada. Caso contrário, são recuperados os dados de entrada que executaram caminhos similares ao pretendido. Estes dados são utilizados como "região" inicial para o processo de busca dos dados que executam o caminho pretendido. É então acionado o algoritmo de busca, cuja função é a de aprimorar as soluções candidatas a resolver o problema (os dados recuperados que executam caminhos similares), fazendo com que estas soluções caminhem em direção à solução final; isto é, a execução do caminho pretendido.

A busca é baseada em um Algoritmo Genético (A.G.) [Gol89]. Este algoritmo tem, neste contexto, a função de combinar e selecionar iterativamente soluções prévias inadequadas com o objetivo de aprimorá-las. O Algoritmo Genético codifica as variáveis de entrada do programa em um "string" de comprimento fixo, referido como "indivíduo". A "população" é um conjunto de indivíduos onde cada um representa uma solução potencial para o problema — referida como *solução candidata.*

Cada indivíduo é avaliado através de uma "função objetivo" que quantifica a qualidade da solução e é utilizada para direcionar o processo de busca realizado pelo Algoritmo Genético. A avaliação é feita utilizando as informações sobre a execução do programa instrumentado com os dados de entrada codificados no indivíduo. A função considera dois fatores: o quanto o caminho executado coincidiu com o pretendido e o erro verificado no predicado no qual o desvio em relação ao caminho pretendido foi observado.

São aplicados os operadores genéticos de seleção proporcional (baseada na função objetivo), recombinação e mutação, e é construída uma nova "geração". Deste modo, a cada geração tem-se a seleção das melhores soluções e a recombinação das mesmas. A busca dá-se em um processo iterativo, no qual as soluções tendem a aproximar-se progressivamente do objetivo, até que ele seja alcançado, isto é, algum dado de entrada que execute o caminho pretendido seja encontrado. A busca pode ser interrompida adicionalmente se um limite pré-definido de custo (número máximo de gerações) for atingido; ou se for identificada a ausência persistente de progresso, possivelmente associada à não executabilidade do caminho pretendido.

Um aspecto importante no modelo proposto é a manutenção da "base de soluções". A cada caminho executado durante o processo de busca da solução — sendo ou não o pretendido, é realizada uma operação de inserção na base, que passa a conter o caminho executado e o respectivo conjunto de valores de entrada que provocou a execução.

Isto permite que sejam usadas informações anteriores do processo de geração de dados quando o testador define um novo caminho a ser executado. Se este caminho não estiver na base (situação na qual a busca torna-se necessária), são recuperados os dados de entrada que executaram caminhos similares ao pretendido. A recuperação leva em conta a *métrica de similaridade de caminhos* definida e tem como objetivo permitir que a população inicial do algoritmo genético contenha soluções cujos dados de entrada estejam próximos dos necessários para a execução do caminho pretendido. Este recurso permite a utilização de informações apuradas dinamicamente sobre o programa (dados de entrada e caminhos executados) para aprimorar a qualidade das soluções iniciais.

Existe uma consonância entre a abordagem de recuperação/adaptação de soluções adotada no modelo da ferramenta e o conceito de "Raciocínio Baseado em Casos" (R.B.C.) [Kol93, AE94, Lea96]. Raciocínio Baseado em Casos é um paradigma de resolução de problemas que utiliza conhecimento específico de situações prévias concretas (casos). Um novo problema é resolvido encontrando um caso passado similar e reusando-o no novo problema. No modelo da ferramenta são recuperados os dados de entrada que provocaram a execução de caminhos similares ao pretendido, apurados em execuções anteriores do programa. Esses dados de entrada são manipulados pelo algoritmo genético. Eles são revistos e adaptados de forma a executar o caminho pretendido. Deste modo é possivel utilizar conhecimento sobre o programa acumulado como um sub-produto do processo de busca. Isto porque a cada avaliação de uma solução candidata tem-se uma execução do programa em teste, que gera informação potencialmente útil quando for necessário gerar dados de entrada para executar um outro caminho do programa.

A Figura 4.1 mostra o diagrama de fluxo de dados da ferramenta no qual podem ser identificados os processos descritos e o relacionamento entre eles.

Nas seções seguintes são detalhados os aspectos conceituais importantes considerados na elaboração deste trabalho. É enfatizado o papel de cada paradigma no contexto da geração de dados de teste e é detalhado o modelo da ferramenta proposta.

4.2 Busca Baseada em Algoritmos Genéticos

Algoritmos Genéticos (A.G.'s) são algoritmos de busca baseados nos mecanismos da evolução, seleção natural e da genética [Gol89, SM94, Zbi96]. Inicialmente propostos por John Holland em 1975 tais algoritmos exploram de forma eficiente informação histórica a fim de investigar novos pontos para a busca que aparentem ser promissores. É utilizada uma população de indivíduos que sofre um processo de seleção na presença de operadores que induzem variações. Uma função de ajuste é utilizada para avaliar os indivíduos e determinar a chance de sucesso na reprodução dos mesmos. Os Algoritmos Genéticos têm sido aplicados em diferentes áreas relacionadas ao aprendizado de máquina e otimização de funções, sendo considerados um método eficaz e robusto de otimização e busca.

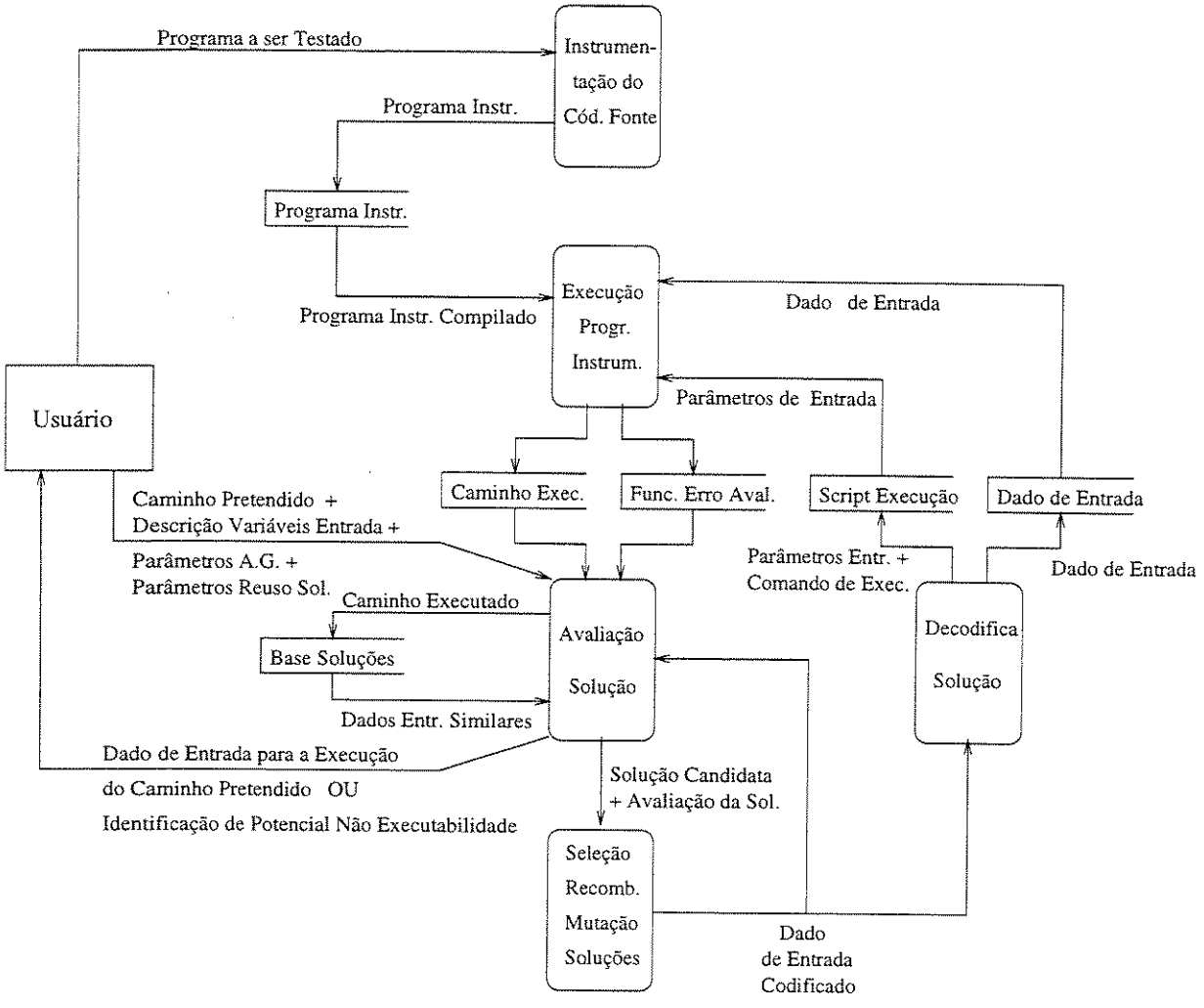

Figura 4.1: Diagrama Geral da Ferramenta de Geração Automática de Dados de Teste

A operação básica dos Algoritmos Genéticos, conceitualmente simples, é descrita a seguir:

1. Mantenha uma população de soluções para o problema;

2. Selecione as melhores soluções para recombinação umas com as outras;

3. Use os descendentes gerados para substituir as piores soluções da população.

4.2.1 Idéias Essenciais

Estes algoritmos refletem o fato de que na natureza a competição por recursos escassos (alimentação, espaço, parceiros para acasalamento) faz com que indivíduos mais bem adaptados dominem os mais fracos. Neste fenômeno denominado "sobrevivência do mais ajustado" baseia-se a seleção natural, elemento essencial na evolução das espécies. Neste mecanismo as características que definem unicamente cada indivíduo determinam a sua capacidade de sobrevivência. Cada característica relaciona-se a um gene. Conjuntos de genes controlando as diversas características formam os cromossomos. A seleção natural enfatiza a sobrevivência dos melhores indivíduos mas também, implicitamente, enfatiza a sobrevivência dos melhores genes. Isto é, indivíduos mais bem adaptados têm mais chances de sobreviver e influenciar gerações posteriores, estendendo suas características genéticas a essas gerações.

A evolução ocorre a partir da existência de uma diversidade de indivíduos competindo entre si e compartilhando material genético na geração de descendentes. A reprodução é caracterizada pela troca de genes entre dois indivíduos da população. Isto possibilita a criação de novas combinações genéticas representadas pelos filhos gerados na reprodução e seus respectivos cromossomos. A reprodução aliada à seleção natural e a mutações genéticas esporádicas, quando realizadas de forma repetida, causam uma evolução contínua no aspecto genético. Tal fato resulta na geração de indivíduos que sobrevivem melhor em um ambiente competitivo por serem melhor adaptados a este ambiente.

Os Algoritmos Genéticos trabalham com populações de potenciais soluções para um problema. Estas soluções são submetidas a um processo de seleção baseado no mérito de cada uma e são aplicados operadores genéticos. Os operadores genéticos manipulam as soluções, transformando-as em um processo aleatório mas, ao mesmo tempo, estruturado. Após selecionadas e possivelmente alteradas as soluções candidatas — os indivíduos da população — constituem uma nova geração. Deste modo ocorre uma exploração de informação histórica para direcionar a busca para novos pontos no domínio que apresentem expectativa de melhorias tendo em vista a solução do problema. Tem-se portanto um procedimento de busca iterativo que pode ser descrito pelo seguinte pseudo-código, o qual refere-se particularmente ao definido por Holland, o "Algoritmo Genético Simples" (A.G.S.) [SM94]:

```
Simple Genetic Algorithm()
 {
  initialize population;
  evaluate population;
  while termination criterion not reached
   {
     select solutions for next population;
     perform crossover and mutation;
     evaluate population;
   }
 }
```

O Algoritmo Genético Simples trabalha com uma população de "strings" binários. Cada "string" composto por "0s" e "1s" é uma versão codificada de uma solução para o problema de otimização. Utilizando os operadores genéticos — recombinação e mutação — o algoritmo cria a geração seguinte a partir dos "strings" da população corrente. Este ciclo é repetido até que a condição de término seja atingida.

O funcionamento dos Algoritmos Genéticos está baseado na idéia de que, com o passar das gerações, as boas soluções tendem a compartilhar partes de seus cromossomos. Essas partes constituem os "esquemas", que são padrões descrevendo subconjuntos de soluções similares, relacionadas a cromossomos com padrões de similiaridade. Esquemas que levam às soluções adequadas são progressivamente identificados e utilizados como "blocos de construção" das novas soluções. O teorema fundamental dos Algoritmos Genéticos diz que esquemas que levam a soluções com alto valor de ajuste, que sejam padrões curtos e de baixa ordem, tendem a crescer exponencialmente nas próximas gerações, enquanto os que levam a soluções com ajuste baixo tendem a diminuir exponencialmente. Deste modo a busca realizada pelos Algoritmos Genéticos pode ser vista como uma competição simultânea entre diversos esquemas, com o objetivo de incrementar a participação de suas instâncias na população. Nesta competição os melhores esquemas tendem a se combinar gerando progressivamente melhores soluções.

4.2.2 Aspectos Importantes nos Algoritmos Genéticos

São aspectos importantes relacionados ao funcionamento dos Algoritmos Genéticos: mecanismo de codificação; função de ajuste e mecanismo de seleção; operadores genéticos e parâmetros de controle.

Codificação

A representação em forma codificada das variáveis envolvidas no problema de otimização é fundamental na estrutura do Algoritmo Genético e influencia criticamente a performance do mesmo. Esta codificação depende da natureza das variáveis e deve mapear a solução para um único "string", comumente em forma binária. Codificações binárias são utilizadas devido à facilidade de implementação e por maximizar o número de esquemas processados [SM94].

Os princípios fundamentais a serem considerados na codificação são referidos como "princípio dos blocos de construção significativos" e "princípio do alfabeto mínimo" [Gol89]. Essencialmente a codificação deve permitir que esquemas curtos e de baixa ordem sejam relevantes na composição de soluções para o problema básico implícito. Deve-se também selecionar o alfabeto mínimo que permita a expressão natural do problema. Seguindo estas diretrizes tona-se possível para o Algoritmo Genético associar altos níveis de ajuste com similaridades entre os "strings" na população.

Em problemas práticos os parâmetros são codificados e mapeados em uma estrutura multi-parâmetros, na qual são associadas faixas de valores e de precisão a cada elemento da estrutura.

Função de Ajuste e Mecanismo de Seleção

A função de ajuste ("fitness function") reflete o objetivo da busca, provendo um mecanismo para avaliar e qualificar cada indivíduo. Conforme a aplicação pode ser adequado normalizar o valor da função objetivo, gerando um ajuste variando entre 0 e 1. Intuitivamente esta função pode ser vista como uma medida de qualidade ou utilidade da solução codificada no indivíduo.

Segundo [Gol89] a função de ajuste deve ser um valor não negativo que reflita o mérito da solução que se deseja maximizar. Caso esta restrição não seja atendida é necessário definir mapeamentos que transformem a função objetivo da busca em uma função de ajuste adequada.

Diversos mecanismos de seleção podem ser utilizados. Este operador genético modela a seleção natural, que enfatiza a sobrevivência dos melhores indivíduos, isto é, os que possuem mais habilidade de sobreviver aos predadores, mais resistência a doenças, dentre outros aspectos.

Um mecanismo amplamente utilizado é a "seleção proporcional". Neste caso soluções com maior ajuste têm uma maior probabilidade de contribuir com um ou mais descendentes na próxima geração. Isto ocorre porque a alocação de descendentes é baseada na razão entre o ajuste do indivíduo e a média de ajuste da população. Ou seja, a chance de o indivíduo ser selecionado é definida pela relação fi/F, sendo fi o ajuste do indivíduo

e F o ajuste médio da população. Deste modo, as soluções melhores, isto é, as que possuem valor de ajuste superior, têm maiores chances de serem selecionadas para formar a próxima geração.

Adicionalmente podem ser adotadas técnicas como "mecanismos de escalonamento" ou "seleção baseada em rank de ajustes" para aperfeiçoar estágios iniciais da busca, quando super indivíduos podem dominar prematuramente a população e estágios finais, quando os melhores indivíduos destacam-se muito pouco em relação à média.

Outra técnica importante refere-se à adoção do "elitismo", que visa preservar os melhores indivíduos da população. Na seleção proporcional não é garantido que a melhor solução para o problema (associada ao melhor indivíduo) sobreviva para a próxima geração. A seleção elitista analisada por DeJong, citado em [Gol89], inclui forçosamente o melhor indivíduo da população na subseqüente. Isto evita que boas soluções sejam perdidas devido ao aspecto aleatório presente na seleção proporcional.

Operadores Genéticos

Os operadores básicos do Algoritmo Genético Simples são: seleção; recombinação e mutação.

É utilizado o operador de *seleção* (descrito anteriormente) que provê um mecanismo para que as melhores soluções sobrevivam. A cada solução é associada uma medida de ajuste que reflete a qualidade da mesma. Quanto maior o valor de ajuste de uma solução da população, maiores são as suas chances de sobreviver e reproduzir, tornando maior a chance desta solução influenciar as próximas gerações.

A operação de *recombinação* é feita após a seleção e simula a troca de material genético entre dois indivíduos da população (resultante da reprodução sexuada). Este mecanismo realiza a troca de partes entre dois "strings" que representam duas soluções candidatas para a solução do problema. Estes "strings" são selecionados da população de forma aleatória e são submetidos à recombinação. A probabilidade de que o operador seja aplicado é um parâmetro de controle do Algoritmo Genético. A idéia da recombinação é que características positivas de dois indivíduos podem ser combinadas, gerando uma descendência cujo nível de ajuste é potencialmente superior ao de ambos os pais [JSE96]. Pode-se dizer que a recombinação (associada à mutação) gera e sobrepõe os "blocos de construção" (esquemas de comprimento curto associados a alto nível de ajuste) para formar indivíduos otimizados [SM94].

Diversos tipos de recombinação têm sido propostos. Na "recombinação de ponto único" (também chamada simples) tem-se a troca de material genético de forma que uma posição p ao longo do "string" é selecionada aleatoriamente. Novos "strings" são gerados pela troca de todos os genes encontrados a partir de p (inclusive) até o final do "string". Outros tipos incluem a "recombinação de dois pontos" ou a "multipontos" as quais seguem

a mesma lógica, exceto pelo número de pontos selecionados e consequentemente o número e o comprimento de segmentos trocados entre os "strings". A "recombinação uniforme" troca bits dos "strings" ao invés de segmentos. Neste caso é feita a troca entre os bits dos dois "strings" de forma probabilística e independente. Outros tipos de recombinação podem ser criados considerando características do problema específico e a codificação utilizada.

A operação de *mutação* é aplicada após a recombinação e consiste em uma alteração ocasional e aleatória (com baixa probabilidade associada) do valor de uma posição do "string" que representa uma solução candidata. Geralmente a mutação de um gene não afeta a probabilidade de mutação de outros genes. Este operador tem a função de restaurar a perda prematura de valores importantes. Isto é, caso em uma população o valor associado a um gene seja o mesmo para todos os indivíduos, a mutação (ao contrário da recombinação) pode gerar um valor diferente para este gene.

Entretanto, os operadores genéticos não se restringem aos mencionados acima. Outros operadores atuando sobre os cromossomos consideram dominância, reordenação de código, inversão, duplicação entre cromossomos, segregação. Operadores orientados à população permitem a exploração de nichos através de operadores como: migração, restrições de casamento e funções de compartilhamento. É possível também utilizar informação dependente do problema para criar novos operadores que exploram conhecimento específico para aprimorar a busca realizada pelo Algoritmo Genético.

Parâmetros de Controle

A busca realizada pelo Algoritmo Genético é caracterizada pela exploração de novas regiões do espaço de busca e a utilização da informação obtida sobre estas regiões para direcionar este processo. O correto balanceamento entre estas duas idéias é feito através da definição de parâmetros de controle pertinentes. Os parâmetros do Algoritmo Genético influenciam de forma significativa o desempenho do mesmo.

A probabilidade de recombinação Pr e a probabilidade de mutação Pm controlam a aplicação dos operadores genéticos. Aumentos no valor de Pr tendem a incrementar a combinação dos blocos de construção, incrementando também a destruição de boas soluções. Incrementos no valor de Pm tendem a transformar a busca genética em uma busca aleatória, entretanto propicia a reintrodução de material genético eventualmente perdido.

O tamanho da população T está associado à diversidade existente entre os indivíduos da população. Um incremento deste parâmetro provoca um aumento da diversidade de soluções, reduzindo a probabilidade do Algoritmo Genético convergir prematuramente para pontos ótimos locais. Por outro lado, tem-se um acréscimo do tempo necessário para que ocorra a convergência para regiões ótimas do espaço [SM94].

Segundo estudos realizados por De Jong envolvendo o uso de Algoritmos Genéticos em otimização de funções (citados em [Gol89]) uma boa performance requer: probabilidede de recombinação alta, probabilidade de mutação baixa, inversamente proporcional ao tamanho da população e tamanho da população moderado. Por outro lado, em [SM94] são destacadas duas possibilidades: baixo tamanho de população associado a probabilidades de recombinação e mutação relativamente altas ou alto tamanho de população com probabilidades de recombinação e mutação menores. Valores típicos no primeiro caso são: $Pr = 0{,}6$, $Pm = 0{,}001$ e $T = 100$. No segundo caso tais valores são: $Pr = 0{,}9$, $Pm = 0{,}01$ e $T = 30$.

4.2.3 Sumário

Algoritmos Genéticos são algoritmos de busca baseados nos mecanismos da evolução, seleção natural e da genética. Os Algoritmos Genéticos trabalham com populações de potenciais soluções para um problema, referidos como indivíduos. Estas soluções são submetidas a um processo de seleção baseada no mérito de cada uma e são aplicados operadores genéticos. Utilizando estes operadores o algoritmo cria a geração seguinte a partir das soluções da população corrente, combinando-as e induzindo mudanças. A combinação e a seleção de soluções quando realizadas de forma iterativa causam uma evolução contínua da população no sentido da solução do problema de otimização.

Em relação a métodos de otimização convencionais, a utilização de Algoritmos Genéticos apresenta as seguintes diferenças:

- Trabalham com parâmetros codificados;
- Realizam a busca com uma população de pontos, ao invés de um único ponto;
- Usam uma função objetivo e não derivadas ou conhecimentos auxiliares;
- Usam regras de transição probabilísticas e não determinísticas.

Identifica-se na seção seguinte a adequação destes algoritmos ao problema da geração de dados de teste. São também descritos trabalhos anteriores relacionando os dois temas.

4.3 Algoritmos Genéticos e Teste de Software

Tendo em vista os conceitos abordados na seção anterior pode-se estabelecer a motivação para a utilização de Algoritmos Genéticos na geração de dados de teste.

Conforme descrito no Capítulo 3 o problema da geração de dados no teste estrutural utilizando a técnica dinâmica pode ser transformado em um problema de otimização.

Considerando a eficácia dos Algoritmos Genéticos para este propósito a adequação é patente. Neste sentido dois pontos merecem destaque:

A busca nos Algoritmos Genéticos é feita com uma população de pontos simultaneamente, explorando diversos picos em paralelo. Isto tende a diminuir a chance de que este processo falhe por encontrar um mínimo (ou máximo) local da função a ser minimizada (ou maximizada) [Gol89]. O método de busca direta chamado de "método da variável alternativa" utilizado por Korel [Kor90] é altamente sensível a funções que apresentam mínimos (ou máximos) locais, podendo falhar nestes casos [MMSC97];

Para que um determinado caminho no programa seja executado é necessário encontrar dados de entrada tais que a *expressão de caminho* seja satisfeita. A expressão de caminho consiste na conjunção dos predicados encontrados ao longo do mesmo e envolve variáveis do programa, constantes, operadores relacionais e lógicos. As variáveis de entrada do programa influenciam os predicados direta ou indiretamente atraves de "cadeias de interações de fluxo de dados" [Nta84]. Deste modo, as expressões de caminho podem ser vistas como expressões aritméticas e lógicas envolvendo as variáveis de entrada do programa. A origem destas espressões na semântica dos programas impede a elaboração de presuposições acerca da natureza das mesmas.

Em suma, devido à potencial diversidade das expressões de caminho não é possível assumir que elas possuam características como: linearidade, continuidade, existência de derivadas, ou que sejam unimodais. Os Algoritmos Genéticos requerem no processo de busca apenas valores que qualifiquem as soluções (o valor de ajuste), não requerendo nenhuma informação adicional, nem fazendo restrições em relação ao tipo de função a ser otimizada.

4.3.1 Trabalhos Anteriores Utilizando Algoritmos Genéticos para a Geração de Dados de Teste

Podem ser identificadas na literatura algumas iniciativas de utilizar algoritmos genéticos para a geração automática de dados de teste de software. O enfoque destes trabalhos é distinto do aqui apresentado sobretudo em relação ao objetivo do teste. Em todos os casos o Algoritmo Genético é utilizado para minimizar erros associados aos predicados a fim de provocar a execução de um ramo específico dos mesmos. Dados são gerados aleatoriamente até que o predicado desejado seja atingido e só então o Algoritmo Genético é aplicado. Apesar do enfoque distinto estes trabalhos enfatizam os aspectos positivos da utilização dos Algoritmos Genéticos no contexto do teste de software.

Jones e outros [JSE96] utilizaram um Algoritmo Genético para gerar dados de teste para o critério "Todos os Ramos". A função de ajuste é baseada no predicado associado

a cada ramo e no número de iterações de laço requeridas. As variáveis de entrada são expressas de forma codificada e concatenada em um "string" de bits. São utilizados operadores genéticos de seleção, recombinação e mutação. A seleção de descendentes é feita de forma aleatória; é aplicada a "recombinação uniforme" e uma mutação cuja probabilidade depende do comprimento do "string" que codifica as variáveis de entrada. São discutidos aspectos importantes da utilização de Algoritmos Genéticos para a geração de dados de teste e é conduzido um experimento para a validação da proposta. Os resultados destacam que a geração utilizando Algoritmos Genéticos requereu uma quantidade de dados de teste substancialmente menor para satisfazer o critério utilizado em relação à geração aleatória. Outro resultado importante diz respeito à boa qualidade dos dados de teste gerados, avaliada utilizando análise de mutantes [DMLS78]. O autor destaca que o algoritmo tende a selecionar dados nos limites dos domínios, o que contribui para a eficácia dos mesmos.

Roper e outros [RMB+95] estabelecem como problema a ser resolvido a geração de dados para atingir um determinado nível de cobertura do código do programa. A aplicação do Algoritmo Genético é feita considerando-se a população do algoritmo como sendo um conjunto de dados de teste de tamanho fixo. Cada indivíduo da população representa um dado de teste, isto é, um conjunto de valores para as variáveis de entrada do programa, utilizado para executá-lo. O ajuste do indivíduo corresponde à cobertura por ele obtida no código do programa. O Algoritmo Genético é utilizado para evoluir a população inicial — gerada aleatoriamente — em direção ao objetivo de atingir a cobertura desejada, neste caso, do critério "Todos os Ramos". Não é considerada nenhuma informação acerca da estrutura interna do programa que é tratado como uma "caixa preta". O autor apresenta um exemplo bastante simples de aplicação da técnica e sugere desenvolvimentos futuros.

Michael e outros [MMSC97, MMS98] convertem o problema de geração de dados de teste no problema de minimização de funções, seguindo a linha proposta por Korel [Kor90]. Assim como nos trabalhos acima citados o critério de teste utilizado é o "Todos os Ramos". O objetivo é portanto cobrir todos os ramos em um programa, sendo que a tentativa de satisfazer uma certa condição é adiada até que sejam encontrados testes que atinjam esta condição. É realizada uma minimização baseada no Algoritmo Genético para cada condição atingida, buscando executar o ramo ainda não executado. A população inicial da busca para uma determinada condição contém os testes que a atingem, podendo também conter valores aleatórios, caso a condição não tenha sido atingida suficientemente. Foi realizado um experimento com um programa escrito em C e contendo 35 pontos de decisão. O objetivo foi o de satisfazer o critério "cobertura de condição-decisão". Foi utilizado o algoritmo de busca com Algoritmo Genético, sendo necessárias 20.581 execuções do programa para atingir 60% de cobertura das condições (cada execução corresponde à avaliação do ajuste de uma solução). Com o mesmo número de execuções e utilizando geração aleatória de dados de teste, foi conseguida a cobertura de 41% da condições.

Neste trabalho também são feitos estudos sobre o efeito da complexidade do programa

no problema da geração de dados de teste. De uma forma geral foi constatado que um aumento da complexidade do problema, avaliada pelo número de variáveis de entrada e pelo número de nós do caminho, faz com que a diferença entre a da geração baseada no Algoritmo Genético e a geração aleatória cresça em termos de custo e eficácia. Isto é, a busca com o Algoritmo Genético mostrou-se mais vantajosa em problemas mais complexos.

A utilização de Algoritmos Genéticos nos trabalhos citados tem a função de minimizar erros associados aos predicados do programa que fazem com que um ramo indesejado seja tomado [JSE96, MMSC97]. Em [RMB+95] o objetivo é maximizar o nível de cobertura do critério Todos os Ramos e são desconsideradas informações sobre o código do programa.

No modelo proposto neste trabalho o Algoritmo Genético tem um papel distinto. O objetivo é maximizar o número de nós executados corretamente em relação ao caminho pretendido e, simultaneamente, minimizar o erro existente no predicado em que o caminho executado passa a diferir do pretendido.

Na seção seguinte é descrita a utilização do Algoritmo Genético na ferramenta de geração de dados de teste desenvolvida.

4.4 Utilização do Algoritmo Genético na Ferramenta Proposta

A ferramenta proposta utiliza a técnica dinâmica de geração de dados de teste. Conforme descrito anteriormente, a técnica dinâmica é baseada na execução real do programa e em métodos de otimização numérica. Diversos trabalhos desenvolvidos nesta linha foram destacados no Capítulo 3.

Neste trabalho utiliza-se o Algoritmo Genético Simples (A.G.S.) como ferramenta de otimização. O papel deste algoritmo é de manipular os valores de entrada do programa a fim de fazer com que o caminho pretendido seja executado. Este propósito, sob a ótica da otimização, pode ser visto como uma busca por dados de entrada que maximizem a proximidade do caminho executado com o pretendido.

Para quantificar a proximidade entre o caminho pretendido e o efetivamente executado estabeleceu-se uma *métrica de similaridade*, que reflete o nível de coincidência entre dois caminhos de programa. Essencialmente a idéia da métrica é a de fornecer uma medida que possa avaliar indiretamente o quão distante um dado de entrada está de executar o caminho pretendido. Como o cálculo direto desta distância não é possível (isto requereria ter a priori o dado de entrada que executa o caminho pretendido, que é a solução do problema) é feita uma aproximação da mesma através da métrica (discutida na Seção 4.6.1 deste capítulo). Esta métrica, juntamente com análise de fluxo de dados, permitem

a utilização de informações dinâmicas da execução do programa para direcionar a busca do Algoritmo Genético.

O indivíduo da população representa de forma codificada um dado de entrada para o programa. Considerando o objetivo de executar caminhos, cada indivíduo é uma solução potencial do problema. Sobre esta população são aplicados os operadores genéticos essenciais do Algoritmo Genético Simples e é feita a seleção das melhores soluções (isto é, as mais próximas de executar o caminho pretendido). Em um processo iterativo, a cada nova geração as combinações de valores para as variáveis de entrada que tendem a executar o caminho pretendido vão sendo exploradas. Tal processo provoca um progressivo incremento da qualidade dos indivíduos da população até que alguma condição de término para busca seja atingida.

O término da busca pode ocorrer nos seguintes casos:

Solução encontrada: Esta situação ocorre quando o caminho executado com os valores de entrada codificados em algum indivíduo coincide exatamente com o caminho pretendido, informado pelo testador.

Número máximo de gerações atingido: É possível que a busca não consiga obter os valores de entrada que executem um caminho pretendido. Nestes casos o limite superior para o número de gerações garante a parada do algoritmo.

Potencial não executabilidade identificada: Caso o caminho pretendido, informado pelo testador, seja não executável, existe a possibilidade que o Algoritmo Genético identifique tal fato. O processo de busca do Algoritmo Genético é monitorado através da apuração e análise de parâmetros que refletem a dinâmica da busca. Esta análise visa identificar, através de uma heurística de caráter dinâmico, situações de ausência persistente de progresso. Tais situações ocorrem quando o Algoritmo Genético encontra soluções que atingem um determinado predicado não factível, o que ocasiona uma *expressão de caminho* não factível. Na Seção 4.7 esta abordagem é detalhada.

A seguir são descritos os principais aspectos do Algoritmo Genético utilizado na ferramenta.

4.4.1 Codificação das Variáveis de Entrada

Como já dito na Seção 4.2 o Algoritmo Genético trabalha com os parâmetros para a busca (as variáveis de entrada no caso) codificados. Para definição da codificação foi necessário restringir os tipos de variáveis tratadas e definir para cada tipo uma forma de mapeamento entre os valores originais (em relação ao tipo) e os codificados. A ferramenta comporta de forma direta o tratamento de variáveis de entrada dos tipos primitivos, isto é inteiros,

reais e caracteres. São também tratados vetores mono-dimensionais destes tipos, o que permite variáveis de entrada do tipo vetor de caracteres (ou "strings").

São variáveis de entrada do programa os valores passados por parâmetros e/ou valores lidos do teclado, além de variáveis de escopo global. A codificação e decodificação destas variáveis é feita a partir do tipo das mesmas. No caso dos tipos inteiro e real (assim como vetores destes elementos) são considerados limites informados pelo testador que determinam faixa de valores e precisão (no caso de variáveis reais).

No caso de variáveis compostas do tipo vetor o testador deve informar o número de elementos da estrutura. O Algoritmo Genético trata internamente cada elemento como uma variável distinta, cujo tipo é o associado ao vetor.

As variáveis de entrada são representadas de forma concatenada e em código binário. Conforme o tipo de cada variável é adotada a codificação respectiva:

Variáveis do tipo inteiro: valores decimais são convertidos para binários. É utilizado um bit de sinal para representação de números negativos. O número de bits utilizados é calculado considerando a informação da faixa de valores tratados pela variável. A decodificação é obtida pela simples conversão binário-decimal.

Variáveis do tipo real: cada variável é mapeada para um inteiro. Isto é feito realizando o produto da variável com uma potência de 10 adequada, considerando a precisão informada da variável. O efeito desta operação é revertido na decodificação da variável, quando é feita a divisão pela mesma potência. Quanto maior a potência utilizada, maior a capacidade do Algoritmo Genético de dircernir e tratar valores deste tipo. Este mecanismo permite a representação de uma variável de entrada tipo real como um valor inteiro;

Variáveis do tipo caráter: serão codificadas a partir do respectivo valor na tabela ASCII. Esta tabela associa caracteres a valores inteiros entre 0 e 127. O valor inteiro será codificado em binário com 7 posições. A decodificação é feita traduzindo-se o valor inteiro binário para decimal e obtendo-se o respectivo caráter associado na tabela ASCII.

Variáveis compostas do tipo vetor: têm cada elemento que compõe a estrutura codificado independentemente, segundo as diretrizes acima.

Utilizando este modelo é construido um "offset" genérico para os indivíduos da população relacionando o vetor de variáveis de entrada ao "string" que representa o cromossomo dos indivíduos.

A operação de codificação não é necessária pois os indivíduos da população inicial são gerados aleatoriamente já codificados, ou são recuperados também na forma codificada da base de soluções (abordada na Seção 4.6).

A operação de decodificação é necessária para que os indivíduos da população sejam avaliados. Esta avaliação é feita através da execução do programa em teste com o dado de entrada codificado em cada indivíduo. Para que isto seja feito é necessário traduzir cada solução candidata codificada para o dado de teste, segundo os tipos originais das variáveis de entrada; são considerados para tanto o "offset" genérico dos indivíduos e o cromossomo de cada indivíduo específico.

Drivers de Compatibilização de Tipos

Estruturas de dados mais complexas como registros, matrizes, ou estruturas dinâmicas, podem ser tratados indiretamente através de *Drivers de Compatibilização de Tipos*. A função destes módulos é a de receber as variáveis de entrada decodificadas nos tipos tratados pelo Algoritmo Genético (descritos acima) e transferi-las para as estruturas complexas aceitas pelo programa em teste. Esta tradução de tipos permite que o Algoritmo Genético realize a busca considerando variáveis primitivas e que o programa receba a entrada segundo alguma estrutura complexa. A seguir exemplifica-se um *driver de compatibilização de tipos* que recebe como entrada nove variáveis do tipo inteiro e chama o programa em teste passando como parâmetro uma estrutura tipo matriz quadrada 3 x 3.

```
void modulo_em_teste(int A[][])
 {
  .
  .
 }
void driver_comp_tipos(int *A[][]);
 {
   int i,j;
   for (i=0; i<=2; i++)
    for (j=0; j<=2; j++)
     scanf("%d\n",&A[i][j]);
 }
main()
 {
   int A[2][2];
   driver_comp_tipos(&A);
   modulo_em_teste(A);
 }
```

Limitações no Tratamento de Variáveis de Entrada

O tratamento dos tipos elementares citados anteriormente associado à utilização de *drivers de compatibilização de tipos* produz um bom nível de flexibilidade à ferramenta; entretanto, algumas questões não são abordadas neste trabalho.

Relacionam-se a seguir algumas situações particulares de difícil tratamento:

Programas com interface não definida estaticamente: Em alguns programas o número de variáveis de entrada pode variar em cada execução. Nestes casos alguma variável determina quantas e quais são necessárias para execução (exemplo: programas de ordenação nos quais o número de elementos não é pré-definido).

Um tratamento possível para este problema é codificar as variáveis considerando limites máximos definidos pelo testador para que a ferramenta gere valores de entrada coerentes, mesmo que redundantes. Exemplo: no caso da ordenação citado é possível gerar como número de elementos para ordenação (primeira variável de entrada) valores entre 0 e 7 e definir 7 variáveis de entrada numéricas referentes aos elementos. Entretanto, a possível redundância de valores tratados pelo Algoritmo Genético tende a dificultar a busca.

Podem também ocorrer situações onde a ordem das variáveis recebidas varia em cada execução. Estas situações não têm como ser tratadas coerentemente na ferramenta;

Estruturas dinâmicas como entrada: Outro problema de difícil tratamento é a geração de dados de teste para programas que manipulam estruturas dinâmicas. Neste caso a execução de um caminho pode requerer uma busca sobre possíveis formas de estruturas dinâmicas submetidas como entrada para o programa. Exemplificando: em algoritmos de busca em árvore binária executar um caminho específico pode requerer gerar como entrada uma árvore com número de elementos e balanceamento específicos;

Programas que manipulam arquivos: Situação análoga ocorre na geração de dados de teste para programas que possuem caminhos cuja execução depende de valores lidos de arquivos. Neste caso o estado do arquivo antes da execução do programa pode ser visto como um dado de entrada adicional que influi na execução do mesmo.

4.4.2 Função de Ajuste

A função de ajuste ("fitness function") reflete o objetivo da busca, provendo um mecanismo para avaliar e qualificar cada indivíduo. Deste modo uma solução da ajuste superior tem maiores mérito e utilidade perante os objetivos estabelecidos.

Como destacado anteriormente a avaliação de cada solução candidata requer a execução do programa com os valores nela codificados. A instrumentação presente no programa em teste — descrita na Seção 4.5 — permite que sejam apurados valores relativos à execução a serem utilizados no cálculo do ajuste.

Diversas funções de ajuste foram analisadas em breves experimentos e selecionadas. Tais experimentos mostraram ser importante a utilização de informações dinâmicas referentes aos fluxos de dados e de controle na qualificação precisa das soluções.

Um dircenimento refinado em relação à qualidade das soluções permite um melhor direcionamento da busca. Isto ocorre porque da qualidade atribuída a uma solução candidata dependerá o nível de influência da mesma nas gerações futuras do Algoritmo Genético, o que de certa forma influencia na escolha da região do domínio a ser explorada nestas gerações futuras.

A função de ajuste selecionada para avaliar cada solução candidata tem a forma:

$$Ft = NC - (\frac{EP}{MEP})$$

sendo que,

Ft: Medida de ajuste da solução candidata;

NC: Número de nós coincidentes do caminho executado em relação ao pretendido. Este valor é computado considerando o número de nós coincidentes entre o caminho executado e o pretendido, a partir do nó de entrada do grafo até o nó onde o caminho executado passa a diferir do pretendido. Esta medida denomidada *Métrica de Similaridade de Caminhos* é discutida detalhadamente na Seção 4.6.1;

EP: Módulo da função de predicado associada ao predicado onde houve o desvio do caminho pretendido. Este valor é obtido a partir de análise de fluxo de dados dinâmica e reflete erro que provocou o desvio;

MEP: Maior valor para a função de predicado apurado dentre os caminhos que executaram o mesmo número de nós corretos. Do mesmo modo que o item anterior é obtido pela análise de fluxo de dados dinâmica.

A computação do valor de Ft é feito através das seguintes etapas:

i) São computados os valores de NC e EP para cada indivíduo da população;

ii) É computado o valor de MEP para cada predicado do caminho pretendido;

iii) Para cada indivíduo da população é calculado o valor $Ft = NC - (\frac{EP}{MEP})$.

Tal função reflete o fato de que uma solução candidata está tanto mais próxima da solução desejada, quanto maior for o número de nós corretos executados. Isto se deve

ao fato de que caminhos que executam a mesma seqüência inicial de nós tendem a estar associados à regiões próximas do domínio de entrada do programa, sendo que esta proximidade é maior, na medida em que for maior o número de nós coincidentes.

Considerando diversas soluções com o mesmo número de nós coincidentes, será considerada mais adequada aquela que apresentar um menor valor absoluto para a *função de predicado* (EP) associada ao predicado onde houve o desvio do caminho pretendido. Esta função reflete erro que provocou o desvio e mede o quão distante está a solução candidata de executar o ramo correto neste predicado.

Deve-se observar que é feita a ponderação do erro verificado no predicado onde houve o desvio do caminho pretendido. Este valor absoluto é dividido pelo maior erro no predicado considerando todas as soluções da população que executaram o mesmo numero de nós corretos, isto é, aquelas que desviaram do caminho pretendido no mesmo predicado. Deste modo o valor de $\frac{EP}{MEP}$ significa um coeficiente de erro da solução candidata perante todas as soluções da população que conseguiram executar o caminho correto até o mesmo predicado do desvio, coeficiente este utilizado como penalidade da solução.

Tem-se portanto uma função de ajuste que atribui maior mérito às soluções que executam um maior número de nós "corretos" e também penaliza aquelas que estão muito distantes de satisfazer algum predicado (considerando a função de predicado). A função de predicado é ponderada de forma que prevaleça o número de nós corretos executados no valor de ajuste. Deste modo o erro no predicado onde o caminho passou a diferir do pretendido representa uma "penalidade" imposta à solução, servindo como um "critério de desempate" para soluções que executam um mesmo número de nós corretos.

Durante o processo de busca cada solução candidata executa um determinado número de nós coincidentes com os do caminho pretendido. O predicado onde houve o desvio em relação ao pretendido é "enxergado" pela solução candidata como um problema de minimização da *função de predicado* deste ramo. Soluções que atingem o mesmo predicado competem na resolução do mesmo problema de minimização, enquanto que as que atingem outros predicados (anteriores ou posteriores no caminho pretendido) competem entre elas na minimização destas outras funções de predicado. Mudanças nos valores de entrada feitos para minimizar as funções de predicado "enxergadas" podem provocar a violação de predicados anteriores. Neste caso esta solução candidata terá sua qualidade diminuida visto que o número de nós coincidentes com o caminho pretendido será menor que o anterior.

Portanto a dinâmica da busca caracteriza-se pela co-existência de dois objetivos: maximizar o número de nós corretamente executados em relação ao caminho pretendido e minimizar a *função de predicado* dos predicados atingidos. Neste sentido a função de ajuste conjuga informação relativa ao fluxo de controle (nós corretamente executados) e informação relativa ao fluxo de dados (conjunto de valores das variáveis que permite a computação da função de predicado) para qualificar cada indivíduo da população.

É descrito a seguir como é feita a computação da função predicado de desvio (valor de *EP*). Na Seção 4.5 aborda-se o modelo de instrumentação do programa em teste, que determina como são obtidas as informações necessárias à computação do valor da função de ajuste para cada solução candidata.

Computação da Função de Predicado

O valor da função de predicado permite dicernir a qualidade de duas soluções candidatas que executaram o mesmo número de nós corretos em relação ao caminho pretendido.

Uma expressão condicional presente em um comando de decisão é referida como *predicado de ramo* (ou apenas *predicado*). *Predicados simples* consistem em expressões relacionais (inequações e equações) do tipo $E1$ *op* $E2$, onde *op* $\in \{<, \leq, >, \geq, =, \neq\}$ e $E1$, $E2$ são expressões aritméticas. *Predicados compostos* são combinações de predicados simples utilizando operadores lógicos dos tipos AND e OR [Kor96, GN97].

Cada predicado simples $E1$ *op* $E2$ pode ser transformado na forma F *rel* 0, onde *rel* $\in \{<, \leq, =, \neq\}$. Esta função é chamada de *função de predicado.* Por exemplo o predicado $a > c$ pode ser transformado em $c - a < 0$. A função F é positiva quando o predicado é falso e negativa quando o predicado é verdadeiro. F é na verdade uma função das variáveis de entrada do programa; portanto, alterações nestas variáveis têm o potencial de influenciar o valor desta função. Porisso, é possível manipular as variáveis de entrada a fim de minimizar o valor de F de um dado predicado. Minimizar F significa caminhar no sentido de satisfazê-lo.

O valor de F, obtido por análise de fluxo de dados dinâmica, classifica as diversas soluções que executam um mesmo número de nós corretos em relação ao caminho pretendido. Menores valores para F significam soluções mais próximas de satisfazer o predicado onde houve o desvio e portanto, mais adequadas.

O cálculo da função de predicado é baseado no tipo de operador relacional envolvido no predicado. A Tabela 4.1 sumariza este cálculo.

Nesta tabela a coluna *Predicado* mostra os possíves tipos de predicados considerando os vários operadores relacionais; *Função de Predicado* é a função F respectiva e *rel* o operador coerente para F *rel* 0.

Em todos os casos o valor de F é o valor *Erro Predicado* (EP) da função de ajuste Ft. F *rel* 0 é a condição necessária e suficiente para o predicado respectivo seja satisfeito.

Especificamente para predicados do tipo $E1 \neq E2$ a condição necessária e suficiente para a satisfação é $F \neq \frac{1}{k1}$, sendo $k1$ um valor numérico menor que 1 (a implementação feita considera $k1 = 0,3$). F reflete o fato de que um aumento no valor de $|E1 - E2|$ tende a gerar melhores soluções, isto é, valores menores para F. Por outro lado, no pior caso tem-se $E1 = E2$ e $F = \frac{1}{k1}$.

Tabela 4.1: Função de predicado segundo o tipo de operador relacional envolvido

Predicado	Função de Predicado	rel
$E1 > E2$	$F = E2 - E1$	$<$
$E1 \geq E2$	$F = E2 - E1$	$\leq$
$E1 < E2$	$F = E1 - E2$	$<$
$E1 \leq E2$	$F = E1 - E2$	$\leq$
$E1 = E2$	$F = \|E1 - E2\|$	$=$
$E1 \neq E2$	$F = \frac{1}{(\|E1-E2\|+k1)}$	$\neq \frac{1}{k1}$
$LG(E1)$	$F = k2$ se $E1 = 0$	
	$F = 0$ se $E1 \neq 0$	
$LG(!E1)$	$F = 0$ se $E1 = 0$	
	$F = k2$ se $E1 \neq 0$	

Predicados lógicos do tipo *if(done)* ou *while(cont)*, sendo *done* e *cont* variáveis do tipo inteiro são referidos na tabela por $LG(E1)$. Para estes predicados $F = k2$, sendo $k2$ um valor inteiro, quando $E1 = 0$. Neste caso o predicado não é satisfeito e $k2$ representa um penalidade fixa para este fato. Quando $E1 \neq 0$ o predicado é satisfeito e portanto $F = 0$. A implementção feita considera $k2 = 100$.

Predicados lógicos do tipo *if(!done)* ou *while(!found)*, sendo *done* e *found* variáveis do tipo inteiro, são referidos na tabela por $LG(!E1)$. Tem-se uma situação oposta à anterior. Nestes casos $F = 0$ quando $E1 = 0$ pois o predicado é satisfeito. Por outro lado, quando $E1 \neq 0$ então $F = k2$. Neste caso o predicado não é satisfeito e $k2$ representa uma penalidade fixa para este fato.

Deve-se observar que o cálculo de F é realizado de forma direta para valores de $E1$ e $E2$ apurados como tipos inteiro e real.

Valores de $E1$ e $E2$ apurados como sendo tipo caráter são tratados pelos respectivos valores da tabela ASCII. Exemplificando, se o predicado for *if(letra == 'd')* e letra (do tipo caráter) $= f$, então tem-se:

$$F = |E1 - E2|$$
$$F = |f - d|$$
$$F = |102 - 100|$$
$$F = 2$$

Esta abordagem permite o tratamento de caracteres como valores inteiros (entre 0 e 127) e a utilização destes valores para a otimização das soluções.

Outro tipo de variável requer um tratamento especial. Predicados envolvendo vetores de caracteres (ou "strings") são permitidos em diversas linguagens. Especificamente na liguagem C são utilizados comandos como: *if (!strcmp(nome1,nome2))* ou *if (!strncmp(nome1,nome2,n))* para para comparar dois "strings" (ambos comandos retornam 0 caso os "strings" sejam idênticos e um inteiro positivo caso contrário). A semântica destes comandos não determina que o valor de retorno seja proporcional à diferença existente entre as duas variáveis. Torna-se necessário portanto atribuir um valor numérico que caracterize a diferença entre dois "strings" e permita o cálculo da *função de predicado* para este tipo de variável. Este valor, referido como *Equivalente numérico (En)*, é definido a seguir:

$$En(string1, string2) = \sum_{i=0}^{n} abs(ascII(string1[i]) - ascII(string2[i]))$$

sendo,

En: Equivalente numérico que caracteriza a diferença entre dois "strings";

n: Número de posições do "string" mais longo;

$ascII(ch)$: Função que retorna o valor na tabela ASCII da variável tipo caracter ch;

$string1$ e $string2$: Variáveis do tipo "string" envolvidas no predicado.

O valor En é proporcional à diferença entre os dois "strings" e assume o valor 0 somente se ambos forem idênticos; portanto, minimizar este valor significa obter "strings" mais "parecidos", enquanto maximizá-lo significa aumentar a diferença entre eles.

O valor En é obtido pela instrumentação do programa em teste (Seção 4.5) e é utilizado para o cálculo do valor da função de predicado considerando a Tabela 4.1. Por exemplo, considerando o comando *if (!strncmp(nome1,nome2,n))*, este cálculo pode ser feito adequadamente identificando o tipo de predicado $E1 = E2$ e fazendo $E1 = En(nome1, nome2)$ e $E2 = 0$.

Este modelo de cálculo do valor *função de predicado* é baseado no trabalho de Korel [Kor90]. Entretanto, foram feitas extensões para o tratamento de variáveis do tipo caráter e "strings" não abordados por este autor, assim como o tratamento de predicados compostos.

A definição da *função de predicado* no caso de predicados compostos é abordada a seguir.

Composição de Funções de Predicado no Caso de Predicados Compostos

Como destacado anteriormente predicados compostos são combinações de predicados simples utilizando operadores lógicos dos tipos AND e OR.

O tratamento deste tipo de predicado requer a definição de como compor o erro resultante quando diversas condições são utilizadas para formar um predicado composto. Em [GN97] tem-se uma abordagem que realiza esta composição considerando o tipo de operador lógico existente no predicado:

[i] $F(c1 AND c2) = F(c1) + F(c2)$;

[ii] $F(c1 OR c2) = MIN(F(c1), F(c2))$.

Portanto, no caso do operador lógico AND a função de predicado resultante é computada realizando-se a soma dos valores da função de predicado associadas a cada condição $c1$ e $c2$. No caso do operador lógico OR a função de predicado resultante considera o menor valor dentre as funções de predicado associadas a cada condição $c1$ e $c2$.

A ferramenta trata predicados compostos da forma descrita acima. Este modelo é compatível com predicados compostos utlizando operadores AND e OR. No entanto, a composição de valores de F quando ambos operadores são utilizadas no mesmo predicado pode gerar valores inconsistentes. Isto porque não é analisada a questão da precedência na aplicação dos mesmos.

Deve-se notar também que no caso de predicados compostos é possível que apenas uma condição seja avaliada como falso ($p1 = falso$ *ou exclusivo* $p2 = falso$) implicando na avaliação de todo o predicado como falso. Neste caso a condição avaliada como verdadeiro não interfere em F. Isto é, $F(p1)$ *ou exclusivo* $F(p2)$ negativa (significando um "acerto" na avaliação da condição) não interfere no valor de F resultante.

4.4.3 Operadores Genéticos

A ferramenta utiliza os operadores genéticos essenciais do Algoritmo Genético Simples (A.G.S.): Seleção, Recombinação e Mutação. Estes operadores são descritos a seguir:

Operador Seleção

Este operador realiza a seleção de indivíduos da população para a composição da próxima geração.

O operador genético de seleção adotado é do tipo *seleção proporcional* no qual a chance de o indivíduo ser selecionado é definida pela relação $\frac{fi}{F}$, sendo fi o ajuste do indivíduo i e

F o valor da média de ajuste população. Deste modo, as soluções melhores, isto é, as que possuem valor de ajuste superior, têm maiores chances de serem selecionadas para formar a próxima geração. Isto faz com que soluções candidatas mais adequadas tendam a ser preservadas, influenciando fortemente as próximas gerações enquanto as menos adequadas tenham a sua influência progressivamente diminuída.

Considerando a função de ajuste descrita anteriormente as melhores soluções são as que executam um maior número de nós corretos em relação ao caminho pretendido. Para cada sub-conjunto de soluções que executaram o mesmo número de nós corretos, são melhores aquelas que possuem menor valor para a função de predicado. Deste modo, valores de entrada associados aos melhores indivíduos tendem a ser preservados para compor novas soluções candidatas nas próximas gerações.

Adotou-se uma técnica próxima do *modelo elitista de seleção* [Gol89]. Em cada geração são identificados os N melhores indivíduos da população, isto é, as N soluções candidatas com maior nível de ajuste. Tais soluções são selecionadas obrigatoriamente, independentemente da seleção proporcional ($N = 2$ na implementação atual). Isto evita a "perda" acidental de boas soluções, desinteressante sob a perspectiva da otimização.

Além disso a adoção deste modelo é adequada ao tratamento da questão da não executabilidade, detalhado na Seção 4.7. O modelo elitista de seleção faz com que o melhor indivíduo de uma dada geração seja também o melhor considerando todo o processo de busca anterior. Quando uma situação de potencial não executabilidade é identificada o melhor idivíduo da população corrente — e portanto o melhor em toda a busca — definirá precisamente o último nó do grafo corretamente executado em relação ao pretendido, que é o último nó anterior ao predicado que deu origem à potencial não executabilidade.

Operador Recombinação

Após a seleção é feita a recombinação (ou "crossover") dos indivíduos. Dentre os indivíduos selecionados alguns são recombinados gerando novas soluções candidatas.

Nesta operação duas soluções candidatas selecionadas aleatoriamente são submetidas à troca de material genético com uma probabilidade Pc — parâmetro do Algoritmo Genético. A recombinação implementada é a chamada "ponto único". Nela é gerado aleatoriamente um ponto p no cromossomo do indivíduo limitado ao comprimento do cromossomo. A recombinação ocorre pela troca do "sub-string" encontrado a partir de p entre os dois indivíduos selecionados.

Nos indivíduos do Algoritmo Genético implementado as variáveis de entrada são representadas de forma concatenada e em código binário, formando o cromossomo do indivíduo. O efeito desta operação portanto é de combinar valores de entrada das duas soluções utilizadas, eventualmente gerando novos valores diferentes dos anteriores.

Operador Mutação

Os indivíduos selecionados e possivelmente recombinados sofrem a ação do operador de mutação.

Na mutação ocorre a alteração aleatória do valor de uma posição do cromossomo do indivíduo. Na codificação binária adotada significa trocar um bit 0 por 1 ou vice-versa. O operador será aplicado com uma probabilidade Pm, informada pelo testador, de forma independente para cada bit.

Como destacado anteriormente a função deste operador é de restaurar a perda prematura de valores importantes. Isto é, caso em uma população o valor associado a um gene seja o mesmo para todos os indivíduos, a mutação (ao contrário da recombinação) pode gerar um valor diferente para este gene. Exemplificando: em uma situação hipotética envolvendo uma variável de entrada do tipo inteiro onde todos as soluções candidatas codificam valores positivos, tem-se a perda do valor 1 para o bit de sinal relativo a esta variável. A mutação pode gerar soluções com esta característica, o que pode ser desejável para execução do caminho pretendido.

4.4.4 Parâmetros de Controle

Conforme descrito na Seção 4.2.2 os parâmetros do Algoritmo Genético determinam um balanceamento entre a exploração de novas regiões do espaço e o direcionamento da busca para regiões do espaço mais promissoras. Todos os parâmetros são fornecidos pelo testador.

A probabilidade de recombinação Pr e a probabilidade de mutação Pm controlam a aplicação destes operadores genéticos. O tamanho da população T determina o número de indivíduos da mesma.

Considerando as referências consultadas [Gol89, SM94] estabeleceram-se faixas de valores para os mesmos:

Taxa de recombinação (Pr): $0,5 \leq Pr \leq 1,0$;

Taxa de mutação (Pm): $0,01 \leq Pm \leq 0,05$;

Tamanho da População(Tp): $20 \leq Tp \leq 100$.

Na aplicação da ferramenta de geração de dados de teste — descrita no Capítulo 5 — foram utilizados os parâmetros:

Probabilidade de Recombinação (Pr) $= 0,8$;

Probabilidade de Mutação (Pm) = 0,02 ou 0,04;

Tamanho da População (Tp) = 40, 60 ou 100;

Em geral, para problemas mais complexos (programas com um maior número de variáveis de entrada e com caminhos pretendidos possuindo um grande número de predicados), optou-se por uma maior diversidade de soluções (Tp maior) e uma maior exploração do domínio de busca (Pm maior), mesmo que em detrimento do desempenho (convergência mais lenta para a solução). O aumento do valor Pr também pode ser utilizado visando uma maior exploração do domínio de busca.

4.4.5 Sumário

Apresentou-se na seção anterior o modelo do algoritmo genético utilizado na ferramenta de geração de dados de teste.

Foram abordados aspectos conceituais importantes relativos aos algoritmos genéticos. Destacou-se a pertinência da aplicação destes algoritmos para o problema da geração de dados no teste estrutural de software e identificaram-se trabalhos anteriores importantes para a definição deste modelo.

O modelo apresentado distingüe-se de trabalhos descritos na Seção 4.3.1 nos seguintes sentidos:

A ferramenta destina-se à geração de dados de teste para a satisfação dos critérios de teste *Potenciais Usos* [MVCJ92]. Para exercitar os elementos requeridos por estes critérios normalmente é necessário executar sub-caminhos no grafo do programa. Deste modo o objetivo na geração de dados não é somente satisfazer um determinado predicado do programa (idéia das abordagens anteriores) mas, além disso, alcançá-lo pela execução de um caminho definido. Deste modo, ao invés de utilizar o algoritmo genético para minimizar erros associados aos predicados, tem-se como objetivo maximizar o número de nós corretamente executados em relação ao caminho pretendido. Considerando a abordagem utilizada isto implica implicitamente em minimizar erros associados aos predicados encontrados ao longo do caminho à medida que as soluções candidatas presentes na população consigam atingi-los.

A medida de ajuste adotada permite a qualificação precisa de todas as soluções presentes na população. Nas abordagens que utilizam apenas o erro no predicado que deseja-se satisfazer, as soluções candidatas que não atingem tal predicado são todas avaliadas igualmente, fazendo com que o processo de busca seja completamente aleatório até que alguma solução atinja o predicado.

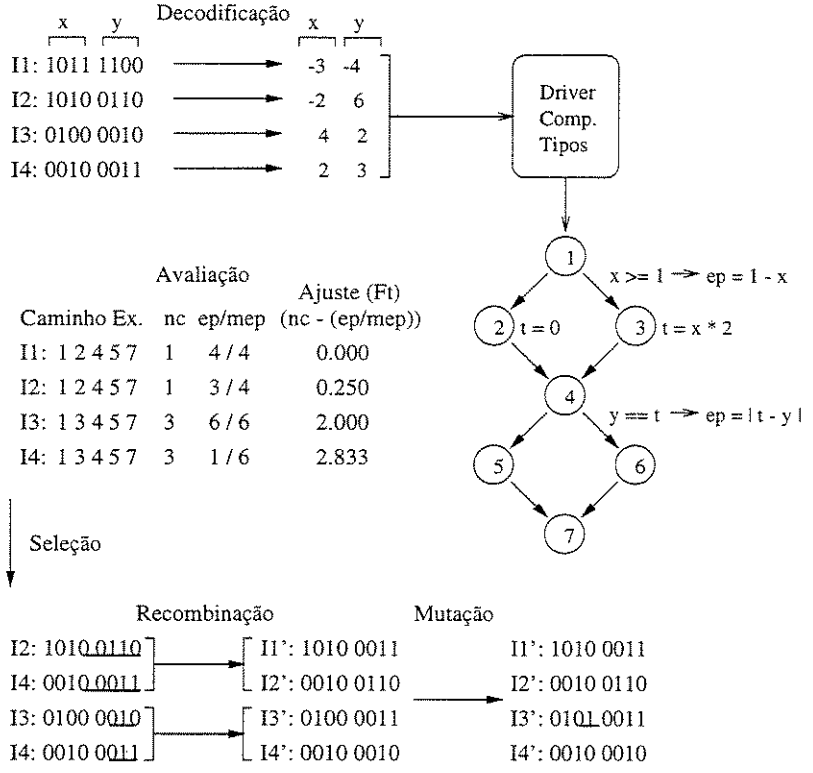

Figura 4.2: Aplicação do Algoritmo Genético na Ferramenta

Portanto, é necessário "contar com a sorte" para encontrar um conjunto de dados de entrada que faça com que o predicado seja alcançado para só então tentar minimizar o erro a ele associado. Isto tende a ser ineficiente quando o objetivo é exercitar algum elemento específico ainda não exercitado no processo de teste, sobretudo em programas mais complexos e predicados difíceis de serem alcançados. Deste modo a utilidade das ferramentas citadas [JSE96, MMSC97] fica restrita se a estratégia de teste for medir a adequação de uma massa de dados pré-existente e complementá-la com dados de teste para exercitar os elementos ainda não executados.

A Figura 4.2 ilustra o processo de geração de dados de teste para executar o caminho pretendido 1 3 4 6 7. São destacados os diversos processos relacionados ao ciclo do algoritmo genético. Os indivíduos existentes na população são *decodificados* obtendo-se as variáveis de entrada do programa nos seus tipos originais (inteiros de -7 à +7). Cada dado de entrada, gerado pela decodificação de um indivíduo, passa (se necessário) pelo *driver de compatibilização de tipos* e é utilizado na execução do programa instrumentado em teste. São obtidas então as informações necessárias ao cálculo do ajuste de cada indivíduo, feito na etapa de *Avaliação*. São aplicados os operadores genéticos na população corrente: *Se-*

leção, Recombinação e *Mutação.* Tem-se assim uma nova população de indivíduos gerada e o início da próxima iteração do ciclo do algoritmo genético. Deve-se notar que nesta figura não foi ilustrado o *elitismo* adotado para a seleção, que preserva obrigatoriamente na nova população as duas soluções candidatas com maior nível de ajuste.

É apresentado na seção seguinte o modelo de instrumentação do programa em teste.

4.5 Modelo de Instrumentação do Programa em Teste

A idéia da instrumentação do programa em teste é transformar o código original em um outro modificado, mas semanticamente equivalente ao original. O programa modificado possui "pontas de prova" com o objetivo de monitorar a execução e coletar informações dinâmicas sobre os fluxos de controle e de dados.

As pontas de prova são basicamente comandos de escrita em arquivo. Elas são inseridas em pontos específicos do código para que as informações coletadas sejam úteis ao processo de geração de dados de teste. O programa fonte original acrescido da instrumentação é referido como *programa instrumentado.*

No contexto deste trabalho pode-se dizer que as informações úteis advindas da execução do programa são aquelas que permitam a avaliação de cada solução candidata presente na população do algoritmo genético. Como descrito na Seção 4.4.2 estas informações dizem respeito aos fluxos reais de controle e de dados.

Deste modo as informações geradas pela execução do programa instrumentado serão utilizadas para direcionar o processo de busca por valores de entrada que executem o caminho pretendido. A instrumentação deve portanto refletir a semântica dos comandos da linguagem e permitir a correta avaliação dos caminhos executados e dos valores associados as expressões envolvidas nos predicados do programa, quando da avaliação dos mesmos.

O modelo de instrumentação adotado considera dois tipos de pontas de prova referidas como: *pontas de prova de caminhos* e *pontas de provas de predicados.* As primeiras têm o objetivo de produzir em arquivo a seqüência de nós executada no programa (caminho executado), enquanto que as *pontas de prova de predicados* visam permitir o cálculo da *função de predicado.*

4.5.1 Monitoramento do Fluxo de Execução

As *pontas de prova de caminhos* seguem o modelo de instrumentação definido em [Mal91] e com aspectos de implementação descritos em [Cha91]. Neste trabalho Chaim detalha a instrumentação associada a cada comando da chamada *Linguagem Intermediária (L.I.).*

Esta linguagem tem como função principal identificar o fluxo de execução em um programa, possuindo comandos do tipo sequencial e de controle de fluxo. Estes comandos são equivalentes aos existentes em linguagens procedimentais: declaração de variáveis; comandos de atribuição; chamadas de procedimentos; comandos de seleção; seleção múltipla; iteração e de transferência condicional. A cada tipo de comando, que representa uma possível estrutura do *grafo de fluxo de controle (G.F.C.)*, é associado um modelo de inserção para as pontas de prova de caminhos.

A ferramenta POKE-TOOL [Cha91] implementa através do script *poketool* a análise estática da linguagem fonte. Dentre outras saídas produzidas é fornecido o programa instrumentado, que contém *pontas de prova* que permitem a obtenção do caminho executado, necessária à geração de dados de teste. A esta instrumentação são acrescidas pontas de prova para monitorar o fluxo de dados, descritas a seguir.

Este modelo é utilizado inteiramente neste trabalho.

4.5.2 Monitoramento do Fluxo de Dados

O objetivo deste monitoramento é o de apurar dados para o cálculo da *função de predicado*. Essencialmente quando tem-se alguma decisão no programa são gravados em arquivo os valores reais envolvidos nesta decisão. Exemplificando: em um predicado do tipo $if(a > b)$ tem-se um comando de escrita em arquivo que grava os valores de a, b e do operador relacional envolvido (>); se houvesse operadores lógicos esta informação também seria gravada.

Naturalmente é essencial garantir que os valores gravados foram efetivamente os valores reais utilizados na avaliação do predicado. Para tanto é necessário que os comandos de escrita sejam inseridos imediatamente antes ou imediatamente depois do predicado. Neste modelo optou-se pela instrumentação após cada decisão. Como será descrito a seguir isto permite que se obtenham informações para o cálculo da *função de predicado*.

A instrumentação de predicados consiste de pontas de prova do seguinte tipo:

$$error\_write(N, E1, E2, OR, OL)$$

Sendo,

error_write: Função que grava no arquivo "error.tes" os parâmetros pertinentes a cada instrumentação;

N: Número do nó no G.F.C. no qual a ponta de prova está inserida;

$E1$: Primeira expressão ou variável presente no predicado;

$E2$: Segunda expressão ou variável presente no predicado;

OR: Operador relacional envolvido;

OL: Operador lógico envolvido no caso de predicados compostos;

Ao parâmetro OR podem ser associados os seguintes valores:

$\{<, \leq, >, \geq, ==, !=\}$: Operadores relacionais aceitos na linguagem C;

PL: Flag para indicar predicado do tipo "lógico positivo", exemplo: *if (done)*;

NL: Flag para indicar predicado do tipo "lógico negativo", exemplo: *if (!done)*;

A ponta de prova é uma chamada para o procedimento que grava em arquivo os valores passados como parâmetros:

```
void error_write(int no, float exp1, float exp2, char *oprel, char *oplog)
 {
   fprintf(error,"%d %f %f %s %s\n", no, exp1, exp2, oprel, oplog);
   nodecount++;
   if (nodecount > MaxNodes) exit(1);
 }
```

Os valores dos parâmetros citados devem ser tais que permitam o cálculo da função de predicado adequada. Isto implica na necessidade de uma correta definição dos operadores envolvidos. A lógica é que a ponta de prova presente em um determinado ponto do programa após uma decisão permite o cálculo da função de predicado associada ao ramo oposto desta decisão. Portanto, quando uma ponta de prova é executada, ela gera no arquivo os dados a serem utilizados caso o ramo tomado na execução não seja o desejado. Isto é, a execução de cada nó do programa situado após uma decisão gera informações que permitem o cálculo da função de predicado referente ao ramo da decisão oposto ao que atinge este nó. A Figura 4.3 ilustra esta idéia.

A Figura 4.3 mostra a inserção de pontas de prova em uma estrutura de seleção do tipo *if-else* e como a informação é utilizada para o cálculo da função de predicado associada a cada ramo. Deve-se notar que se o ramo $(1,3)$ associado ao predicado $(i > j)$ for executado a ponta de prova do nó 3 gravará em arquivo a tupla $3, i, j, <=, \#$. Caso o fluxo de execução desejado neste caso fosse pelo ramo $(1,2)$ esta tupla permitiria o cálculo da função de predicado associada ao ramo $(1,2)$, a qual deveria ser minimizada para que este ramo seja tomado (o que implicitamente acontece com a maximização do valor de ajuste, vide Seção 4.4.2). Isto procede pois quando o ramo $(1,2)$ é o pretendido na execução ocorrerá um "erro" quando $(1,3)$ for executado. Portanto, em $(1,3)$ será computado o erro (*função de predicado*) referente a $(1,2)$ e vice-versa.

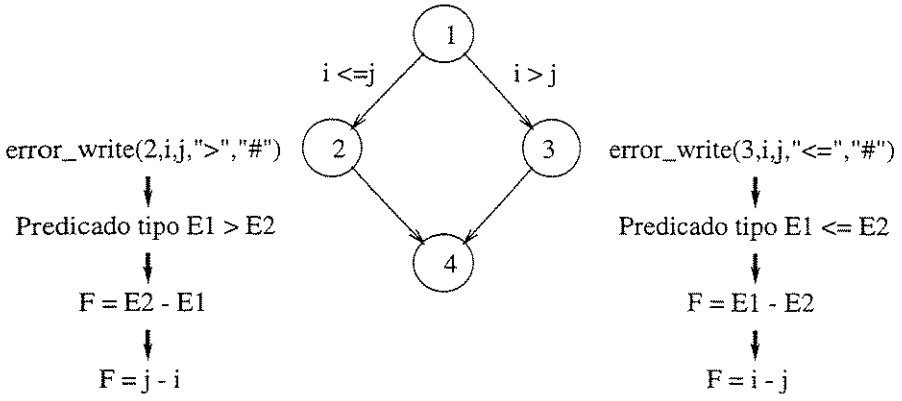

Figura 4.3: Modelo de Instrumentação e Cálculo da Função de Predicado

Instrumentação de Predicados Compostos

O parâmetro OL permite o tratamento de predicados envolvendo os operadores lógicos AND *e* OR representados, respectivamente, por && e ||. Para que isto seja possível é necessário explicitar através de pontas de prova as condições do tipo $E1$ *op* $E2$, onde $op \in \{<, \leq, >, \geq, =, \neq\}$, assim como o operador lógico envolvidos nos predicados.

O trecho de programa da Figura 4.4 ilustra a utilização da instrumentação no caso de predicados compostos.

Neste caso cada chamada da função *error_write* grava informação relativa a uma condição do predicado. Note-se que foram inseridos os parâmetros OL que indicam o operador lógico utilizado no predicado. As definições dos operadores relacionais e lógicos devem ser coerentes a fim de possibilitar o cálculo da função de predicado. No exemplo anterior a instrumentação dos nós 2 e 3 reflete o caráter complementar da condição necessária para que sejam tomados o *if* e o *else*, respectivamente.

Modelo de Instrumentação Para Comandos de Seleção e de Repetição

A Figura 4.5 ilustra a inserção de pontas de provas de predicados no caso de comandos de repetição. Nesta figura as pontas de prova são representadas por $PP1$ e $PP2$ e os nós de entrada dos comandos por Ne. $n1$ e $n2$ representam comandos presentes nos nós distintos pertencentes a cada estrutura e pr representa os predicados existentes nas mesmas.

A Figura 4.6 ilustra a inserção de pontas de provas de predicados no caso de comandos de Seleção. Valem as mesmas convenções da Figura 4.5, ns representa o nó de saída dos comandos.

```
if ( ( i > j ) && ( a <= b ) )
 {
    error_write(2,i,j,"<=","||");
    error_write(2,a,b,">","#");
    .
    .
    .
 }
else
 {
    error_write(3,i,j,">","&&");
    error_write(3,a,b,"<=","#");
    .
    .
    .
 }
```

Figura 4.4: Instrumentação no Caso de Predicados Compostos

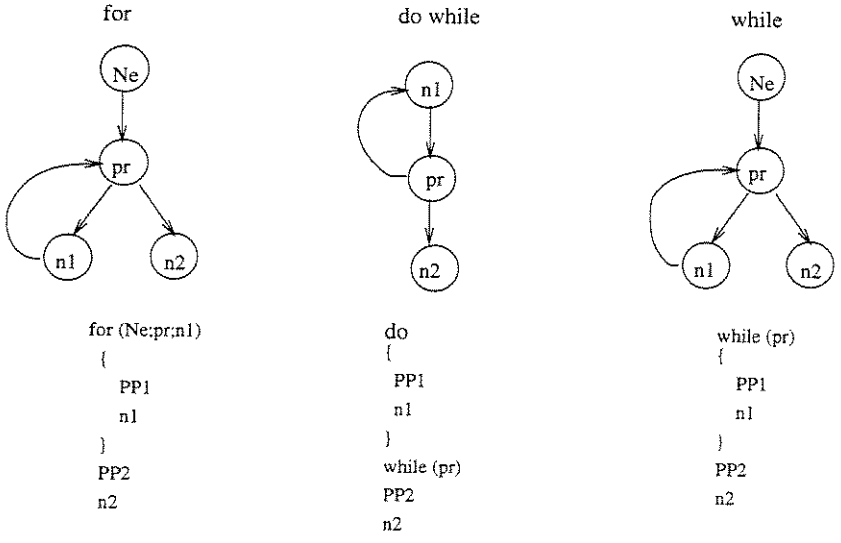

Figura 4.5: Modelo de Instrumentação Para Comandos de Repetição

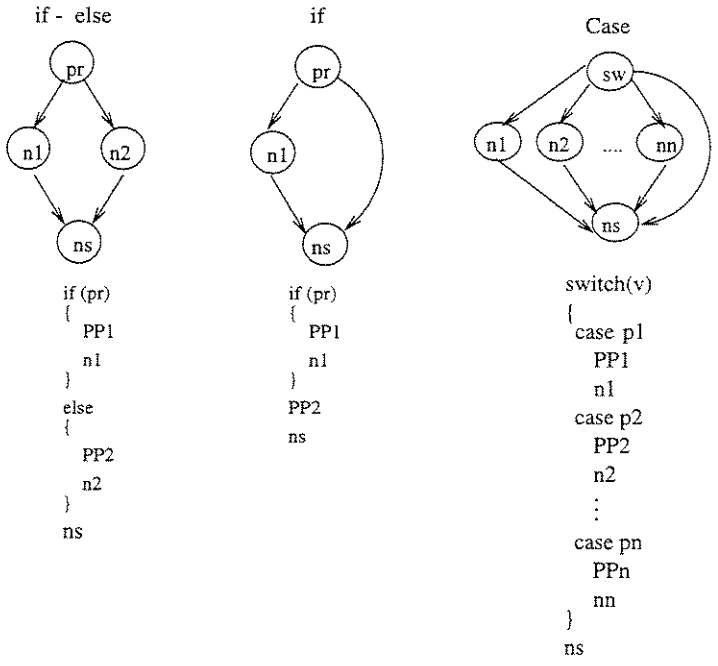

Figura 4.6: Modelo de Instrumentação Para Comandos de Seleção

Tratamento de Comandos Tipo *case*

Merece atenção especial a instrumentação de comandos do tipo *case*. Neste caso a lógica ilustrada na Figura 4.3 não se aplica. Como existem diversas alternativas não é possível pre-determinar qual função de erro deverá ser computada em cada opção do case. Em relação a este problema podem ser utilizadas três abordagens distintas:

Realizar uma instrumentação "dependente do caminho pretendido". Neste caso, em cada opção do case deve ser inserida uma ponta de prova que permita a computação da função de predicado referente ao predicado pretendido. Seguindo a lógica da Figura 4.3 identifica-se o ramo pretendido no case (ramo que deseja-se executar) e inserem-se nos nós do case associados aos demais ramos (que não o pretendido) pontas de prova adequadas para o cálculo da função de predicado referente ao ramo pretendido. Deste modo, independentemente do ramo realmente executado nesta estrutura, ter-se-á a correta computação da função de predicado referente ao ramo pretendido;

Inserir pontas de provas "nulas" do tipo $error\_write(N, 0, 0, "\#", "\#")$ onde N é o número do nó no G.F.C.. Isto permite que o correto funcionamento do gerador de dados; entretanto, a busca por valores de entrada que executem o ramo preten-

dido do case será não direcionada e, portanto, menos eficiente;

Uma outra alternativa adotada por diversos autores que utilizam a técnica dinâmica de geração é exigir que estruturas do tipo case sejam transformadas em estruturas do tipo if-else [Kor90, FK96, GN97]. Outros autores não fazem referência ao tratamento dispensado neste caso [MMSC97, JSE96, PH97, GMS98].

Tratamento de *"strings"*

Conforme destacado na Seção 4.4.2, para o cálculo da *função de predicado* no caso de predicados envolvendo "strings", é necessária a computação do valor *equivalente numérico*. Este valor caracteriza a diferença entre dois "strings" e é computado pela seguinte função, inserida no programa instrumentado:

```
int en(char str1[], char str2[])  /* instrumentacao para o tratamento de strings */
{
 int i = 0;
 int err = 0;
 while  ( (str1[i] != '\0') || (str2[i] != '\0') )
 {
  err = err + (int) fabs( (int) (str1[i] - str2[i]) );
  i++;
 }
 return(err);
}
```

Nestes casos as pontas de prova devem ser inseridas conforme a lógica descrita pela Figura 4.3. Exemplificando: *error_write(9,en(nom,"test"),0,"==","#")* permitiria o cálculo da *função de predicado* visando aproximar o "string" *nom* do conteúdo *"test"*.

Chamadas de Função Presentes em Predicados

Quando ocorrem chamadas de função presentes em predicados é necessário um cuidado especial para que a instrumentação não altere a semântica do programa. Considere o seguinte trecho de programa:

```
...
if (func(x,y))
 {
  error_write(2,func(x,y),0,"NL","#")
  ...
 }
...
```

Neste exemplo a dupla invocação da função *func* pode alterar a semântica do programa. Um possível tratamento seria criar automaticamente um variável temporária, do tipo retornado pela função, e utilizá-la de forma que o número de invocações de *func* não seja alterado em relação ao programa original:

```
...
temp1 = func(x,y);
if (temp1)
 {
  error_write(2,temp1,0,"NL","#")
  ...
 }
...
```

Este tratamento pode ser aplicado também para predicados que fazem definições de variáveis, por exmplo: $if(cont--)\{...\}$, situação na qual a variável *cont* é utilizada no predicado e então decrementada.

4.5.3 Tratamento de Laços

As pontas de prova de predicados são adicionadas também em comandos seqüenciais que não estão antes de decisões; isto faz com que cada nó do G.F.C. contenha pelo menos uma ponta de prova deste tipo. Neste caso são inseridos comandos: *error_write (N,0,0,"#","#")*, onde N é o número do nó no G.F.C.. Esta restrição é estabelecida como um recurso para facilitar o cálculo da *função de predicado*. Tem-se deste modo no arquivo *error.tes* a sequência de nós executada no programa, o que facilita a localização de qual valor no arquivo deve ser considerado. Isto é particularmente importante quando o desvio em relação ao pretendido ocorre em nós inseridos em laços. Neste caso, além de identificar o nó de desvio, é necessário também identificar em qual iteração do(s) laço(s) ele ocorreu.

Como o programa instrumentado será executado sob o comando da ferramenta de geração de dados é desejável um maior controle sobre a execução, especialmente em relação aos laços. Para tanto a instrumentação cria uma variável do tipo inteiro *nodecount*, incrementada a cada chamada de *error_write*. Esta variável, após cada incremento, tem o seu valor testado em relação a um limite *MaxNodes* (na implementação atual trata-se de uma constante: $MaxNodes = 10.000$). Se o limite é atingido a execução do programa instrumentado é finalizada. Este tratamento visa impedir que a geração de dados seja prejudicada quando os valores atribuidos às variáveis de entrada provoquem *laços infinitos*.

4.6 Reuso de Soluções Passadas Baseado em Analogia

Um aspecto importante no modelo proposto é a manutenção de uma "base de soluções". Esta base contém informações dinâmicas sobre o teste de um dado programa. O principal objetivo é obter uma redução do custo da busca no processo de geração de dados.

Durante o processo de geração de dados de teste informações relacionadas à execução do programa são guardadas. Em cada execução do programa em teste são inseridos nesta base: o caminho completo executado no grafo de fluxo de controle do programa e o respectivo conjunto de valores (codificados) para as variáveis de entrada — referido como dado de entrada.

Esta informação, que relaciona valores de entrada e fluxo real de execução, pode ser vista como um importante conhecimento agregado sobre o programa em teste. Conhecimento de natureza dinâmica, ao contrário do gerado através da análise estática do programa, realizada para a geração dos elementos requeridos pelos critérios de teste. A base de soluções tende a crescer continuamente durante o processo de geração de dados, contendo cada vez mais informação.

A informação contida na base relaciona-se a um aspecto primordial da geração de dados de teste: ela reflete como ocorre o mapeamento entre os componentes estruturais (caminhos do programa) e conjuntos de valores de entrada que os exercitam. A utilização deste conhecimento acumulado dá-se em situações posteriores no processo de teste, quando o objetivo for o de exercitar um novo elemento requerido. Nesta ocasião informações dinâmicas anteriores sobre o programa são úteis na definição de qual região do domínio de entrada deve ser explorada na busca pelos dados que executem o novo elemento requerido. Isto é possível pois as execuções anteriores geraram um conhecimento sobre a relação entre características dos dados de entrada (faixa e combinação de valores de entrada) e partes do código exercitadas (caminho executado).

Portanto, a cada caminho executado durante o processo de busca da solução — sendo ou não o pretendido, é realizada uma operação de inserção na base, que passa a conter o caminho executado e o respectivo conjunto de valores de entrada que provocou a execução.

Quando o testador define um novo caminho pretendido é verificado se este caminho existe na base de soluções, isto é, se em algum momento anterior no processo de geração de dados este caminho, agora pretendido, foi fortuitamente executado. Caso positivo, o conjunto de valores de entrada respectivo é recuperado e fornecido ao testador. Caso contrário, são recuperados os dados de entrada que executaram caminhos similares ao pretendido.

A recuperação de caminhos similares tem como objetivo permitir que a população inicial do algoritmo de busca contenha soluções cujos dados de entrada estejam próximos

dos necessários para a execução do caminho pretendido. Para tanto foi definida uma métrica de similaridade de caminhos de programa.

4.6.1 Métrica de Similaridade de Caminhos

A métrica de similaridade adotada reflete o nível de coincidência entre dois caminhos de programa. Na recuperação dos caminhos da base esta medida será aplicada para quantificar a similaridade entre o caminho pretendido e cada caminho existente na base.

A cada caminho pertencente à base de soluções é associado um valor denominado *medida de similaridade*. Este valor é igual ao número de nós coincidentes entre o caminho executado e o pretendido, a partir do nó de entrada do grafo até o nó onde o caminho executado passa a diferir do pretendido. Portanto, a medida de similaridade pode variar entre 1 e o número de nós do caminho pretendido. No primeiro caso (similaridade = 1) apenas o nó de entrada do grafo é comum aos dois caminhos; no segundo caso (similaridade = número de nós do caminho pretendido) o caminho existente na base e o pretendido são idênticos.

Por exemplo, em relação a um caminho pretendido hipotético *1 3 4 5 7 9 11* um caminho executado *1 3 6 9 11* teria similaridade 2 (nós 1 e 3 coincidentes), enquanto o caminho *1 3 4 5 8 10 11* teria similaridade 4 (nós coincidentes 1, 3, 4 e 5). Esta medida de similaridade entre caminhos foi definida analisando-se a forma como os predicados do programa subdividem o domínio de entrada em regiões.

A idéia essencial da métrica de similaridade, considerando que para executar um determinado caminho é necessário satisfazer N predicados, é a seguinte:

Um dado de entrada *D1* satisfaz os primeiros *M1* predicados do caminho ($M1 < N$) e um dado de entrada *D2* satisfaz os primeiros *M2* predicados do caminho ($M2 < N$); se ($M1 > M2$) então *D1* tende a estar mais próximo do conjunto de dados que satisfaz os N predicados (e executa o caminho) do que *D2*.

Isto significa que a região do domínio de entrada que contém dados que satisfazem todos os N predicados pode ser vista como uma sub-região específica contida na região do domínio de entrada formada por dados de entrada que satisfazem apenas os primeiros *M1* ou *M2* predicados do caminho. Desta forma, iniciar o processo de busca com algumas soluções que executaram caminhos similares ao pretendido — segundo a métrica apresentada — significa iniciar a busca com uma população de soluções "próximas" da região do domínio que contém dados que executam o caminho pretendido, o que tende a minimizar o custo deste processo.

Vale destacar que durante execuções distintas do programa que atingem um mesmo predicado P, pode-se ter, do ponto de vista dinâmico, P representando diferentes restrições quando visto em função das variáveis de entrada. Assim, do ponto de vista

dinâmico, a equivalência do predicado P em diferentes execuções só é garantida caso o caminho executado até P coincida nestas diversas execuções; isto sem considerar o efeito de variáveis compostas e ponteiros, que restringem ainda mais a possível equivalência. Se variáveis desses tipos interferem em P este predicado será equivalente em diferentes execuções somente se os valores das variáveis de entrada forem idênticos nessas execuções. Estes aspectos refletem-se na métrica de similaridade visto que ela considera o número de nós coincidentes a partir do nó de entrada do grafo e não apenas a coincidência de predicados em diferentes caminhos.

A utilização da idéia de similaridade de caminhos relaciona-se a um procedimento comumente realizado de forma intuitiva pelo testador. Para gerar um novo dado de teste que exercite um determinado elemento requerido pode-se utilizar um dado anterior que executou regiões próximas no código e tentar transformá-lo analisando o código e utilizando bom senso. Isto é, pode-se usar dados de entrada semelhantes para exercitar partes "próximas" no código.

Quando é definido um um novo caminho pretendido para a geração de dados de teste é feita uma operação de recuperação de caminhos similares. São recuperados primeiro os dados de entrada associados aos caminhos com maior nível de similaridade; em seguida, os associados a caminhos com níveis sucessivamente menores. Estes dados de entrada recuperados vão constituir a população inicial do algoritmo genético.

A base de soluções é um arquivo no qual cada linha contém a tupla: caminho completo executado (seqüência de nós) e conjunto de valores de entrada (em forma codificada, tal qual manipulada pelo algoritmo genético).

4.6.2 Raciocínio Baseado em Casos

Existe uma consonância entre a abordagem de recuperação/adaptação de soluções adotada no modelo da ferramenta e o conceito de "Raciocínio Baseado em Casos" (R.B.C.) [Kol93, AE94, Lea96].

Raciocínio Baseado em Casos é um paradigma para resolução de problemas em que as soluções são obtidas através da identificação de casos passados similares e do reuso destes casos no novo problema. O Raciocínio Baseado em Casos pode também ser visto como uma abordagem de aprendizado incremental. Isto porque novas experiências são retidas sempre que um novo problema é resolvido. Segundo os autores citados, raciocinar reusando casos passados é uma maneira poderosa e freqüentemente aplicada por humanos para solução de problemas, podendo ser considerada uma forma de "analogia intra-domínio".

No Raciocínio Baseado em Casos a principal fonte de conhecimento consiste de uma memória de casos relacionados a episódios específicos anteriores e não de regras gerais. Os casos podem ser vistos como pares problema-solução respectiva. Segundo Kolonder

e Leake [Lea96] um caso é "uma porção de conhecimento contextualizado representando uma experiência que ensina uma lição fundamental para atingir os objetivos do raciocínio". Em suma um caso tem o potencial de permitir que um objetivo, ou conjunto de objetivos, seja atingido mais facilmente no futuro.

Portanto, no Raciocínio Baseado em Casos o raciocínio é baseado na lembrança, isto é, novas soluções são geradas pela recuperação de casos relevantes da memória e pela adaptação destes casos às novas situações. Esta abordagem é baseada em dois fatos relacionados a natureza do mundo: O primeiro fato é que o mundo é regular, isto é, problemas similares possuem freqüentemente soluções similares. O segundo é que tipos de problemas e de agentes tendem a se repetir, ou seja, problemas futuros tendem a ser semelhantes aos problemas correntes. Em domínios nos quais estes dois fatos são identificados é compensador recuperar e reusar raciocínios e/ou soluções anteriores.

Outro aspecto importante no Raciocínio Baseado em Casos é o aprendizado pela experiência. O conhecimento está constantemente mudando na medida em que novas experiências geram novos casos que são guardados para uso futuro. Portanto, o apredizado pode ser visto como um sub-produto natural do processo de resolução de problemas.

Um benefício interessante é que o reuso de soluções anteriores ajuda a incrementar a eficiência na resolução de problemas. Isto ocorre devido à possibilidade de recuperar raciocínios anteriores ao invés de repetir o esforço anterior, o que é particularmente interessante em problemas complexos, cuja resolução tende a ser custosa e difícil.

Em geral as tarefas dos sistemas que utilizam Raciocínio Baseado em Casos são relacionadas a duas classes distintas: interpretação de problemas e resolução de problemas. Na interpretação são utilizados casos passados para classificar novas situações através da comparação com situações já classificadas. Na resolução de problemas são utilizados casos anteriores como possíveis soluções para aplicação nas novas circunstâncias. Este processo envolve avaliação da situação, recuperação de casos e avaliação de similaridade.

De forma genérica os problemas são abordados do seguinte modo: dado um novo problema o sistema baseado em casos realiza uma avaliação da situação e gera uma descrição do problema. É feita uma busca por problemas anteriores com descrições de problema relevantes (isto é, similares). A solução do problema mais relevante é utilizada como ponto de partida para a geração da solução para o novo problema. A solução gerada é guardada para futuro reuso.

Ciclo do Raciocínio Baseado em Casos

Existe uma natureza cíclica no Raciocínio Baseado em Casos [Lea96] destacada na Figura 4.7 [AE94], onde é também possível identificar tarefas principais relacionadas ao paradigma:

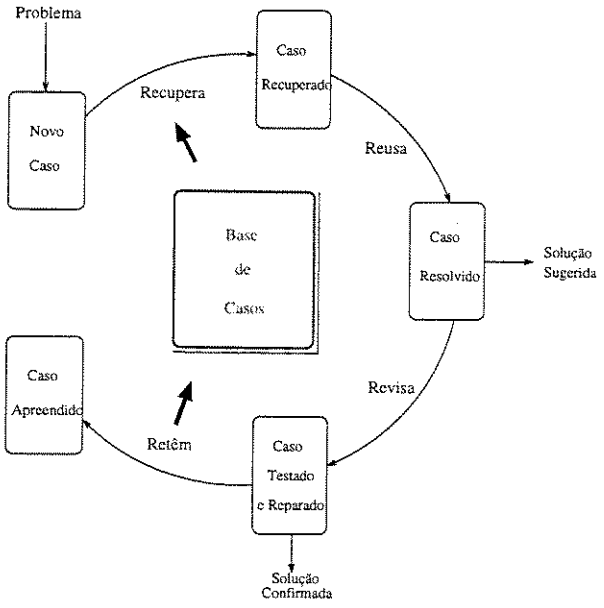

Figura 4.7: Ciclo do Raciocínio Baseado em Casos

Recuperação do(s) caso(s) similar(es): O propósito é selecionar casos que possam ser adaptados facilmente ao problema corrente, isto é, casos selecionados devem ter soluções próximas da do problema corrente. A medida de similaridade funciona como uma aproximação "a priori" de reusabilidade e adaptabilidade das soluções. Esta medida pode considerar cálculos baseados em características locais, globais, inferências lógicas ou ainda em comparações de grafos. Essencialmente a recuperação de casos é uma busca realizada na base considerando índices a eles associados. Os índices refletem características dos casos que os distigüem uns dos outros. A nova situação é utilizada como uma chave de recuperação de situações similares;

Reuso da informação e conhecimento neste(s) caso(s) para resolver o problema: Envolve a aplicação das soluções associadas aos casos recuperados no problema corrente. Isto é, ocorre a aplicação da solução sugerida ao problema real, etapa realizada fora do sistema de Raciocínio Baseado em Casos. Eventualmente são necessárias adaptações para que a aplicação seja possível;

Revisão da solução proposta e reparo/adaptação da mesma: Consiste na avaliação da aplicação das soluções recuperadas ao problema real e eventual adaptação das mesmas. Como situações passadas não são sempre iguais às novas, adapações são necessárias. Os métodos para adaptação são classificados como: baseados em substituição, transformação, derivacional ou de propósito especial [Kol93];

Retenção da solução caso ela seja útil para solução de problemas futuros: É essencial para que ocorra o processo de aprendizado. O sistema de Raciocínio Baseado em Casos tende a tornar-se mais competente com o tempo, gerando cada vez melhores respostas do que antes. Novos casos fornecem contextos familiares adicionais para resolver novos problemas e avaliar novas situações. O aprendizado também pode tornar possível antecipar e evitar erros feitos no passado.

Adaptação de Casos

A *adaptação de casos* é um aspecto central do Raciocínio Baseado em Casos que ainda permanece em aberto [Lea96]. Este processo envolve adaptar os casos recuperados para que eles se ajustem às novas circunstâncias, o que é essencial para a habilidade do sistema em resolver novos problemas e reparar soluções que falharam.

Métodos para adaptaçãao *baseados em substituição* [Kol93] visam trocar valores adequados à nova situação em lugar dos recuperados. Nesta abordagem destacam-se: o *ajuste de parâmetros*, uma heurística para ajustar parâmetros numéricos que relaciona variações de entrada e efeitos produzidos na solução. A *busca local* procura em uma estrutura de conhecimento auxiliar maneiras de identificar e mudar valores incompatíveis com a nova situação. *Substituição baseada em casos* usa outros casos para sugerir substituições;

Métodos *baseados em trasformação* transformam uma solução antiga em uma que funcione em uma nova situação. São utilizadas *transformações de senso comum*, isto é, heurísticas são aplicadas para trocar, remover ou adicionar componentes em uma solução. Podem também ser aplicadas *reparações guiadas por modelo* que partem de um modelo causal para sugerir reparos.

Adaptações de propósito especial abordam adaptações não tratadas por outros métodos. São utilizadas heurísticas específicas do domínio para fazer reparos, comumente através de sistemas baseados em regras.

Sumário

O Raciocínio Baseado em Casos trata de forma pragmática situações que envolvem a resolução de problemas. Neste paradigma novos problemas são resolvidos através da recuperação e adaptação de soluções passadas que foram utilizadas em problemas análogos. A cada novo problema resolvido agrega-se conhecimento potencialmente útil em situações futuras.

A relação existente entre o modelo da ferramenta e o raciocínio baseado em casos é enfatizado na seção seguinte.

4.6.3 Reuso de Soluções Passadas e Raciocínio Baseado em Casos

O modelo proposto para a ferramenta e, em particular, o reuso de soluções passadas no problema de geração de dados de teste apresenta uma forte correlação com paradigma Raciocínio Baseado em Casos. Tal correlação pode ser observada ao se analisar os processos envolvidos no reuso de soluções: gravação na base de soluções de caminhos executados e respectivos dados de entrada; recuperação de dados de entrada utilizando a Métrica de Similaridade de Caminhos e revisão/adaptação de soluções utilizando Algoritmo Genético.

A relação entre os processos do modelo de reuso de soluções e as tarefas presentes no ciclo que caracteriza o Raciocínio Baseado em Casos é ilustrada na Figura 4.8 e destacada a seguir:

Recuperação dos casos similares: Para execução de um novo caminho são recuperados da base de soluções os dados de entrada que executam caminhos similares, considerando a métrica de similaridade definida. A tupla [caminho, dado respectivo] representa um caso. Isto permite recuperar o dado de entrada que executa o caminho pretendido caso ele esteja na base, situação comumente verificada. Caso este caminho não seja encontrado a recuperação de caminhos similares permite iniciar a busca com uma população de soluções "próximas" da região do domínio que contêm dados que executam o caminho pretendido, o que tende a minimizar o custo deste processo.

A fim de manter uma boa variabilidade na população inicial é limitada a recuperação de soluções. O valor máximo de soluções recuperadas é igual à metade do tamanho da população do Algoritmo Genético. Deste modo espera-se conciliar a variabilidade essencial para o sucesso da busca com o ganho de performance devido à recuperação de dados que executam caminhos similares.

Reuso da informação e conhecimento neste(s) caso(s) para resolver o problema: Os dados recuperados são utilizados como solução do problema (caso o caminho pretendido esteja na base de soluções) ou como população inicial para o algoritmo de busca. O reuso é feito executando-se o programa com os dados de entrada recuperados e monitorando o caminho executado;

Revisão da solução proposta e reparo/adaptação da mesma: Caso a solução não seja encontrada na base de soluções, o conjunto de dados recuperados são submetidos ao algoritmo de busca. O algoritmo genético desempenha a função de avaliar as soluções recuperadas (que passam a constituir a população inicial do Algoritmo Genético) e adaptá-las a fim de que o caminho pretendido seja executado. Este algoritmo manipula as soluções candidatas presentes na população inicial através da aplicação dos operadores genéticos;

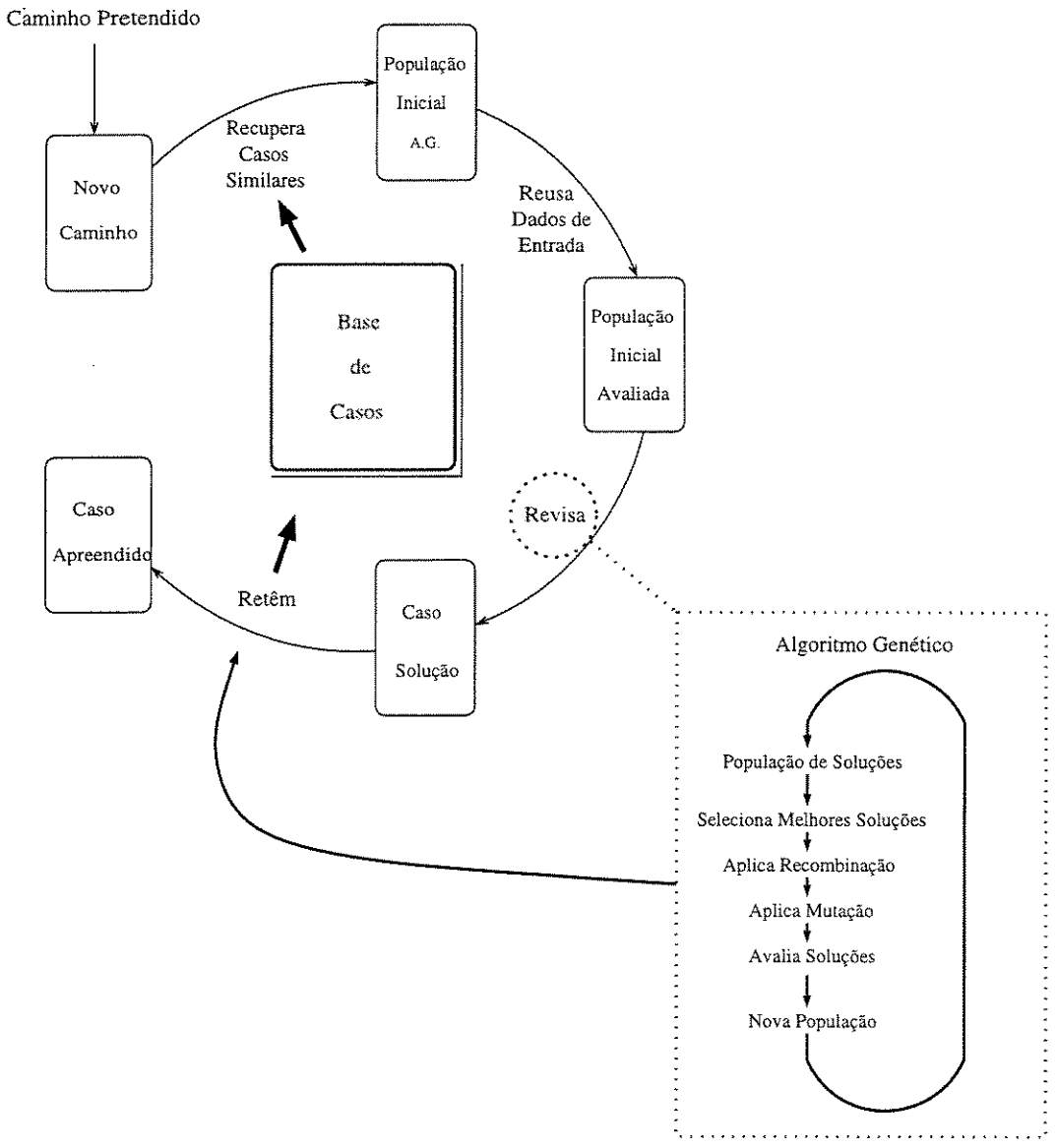

Figura 4.8: Raciocínio Baseado em Casos no Contexto da Ferramenta

Retenção da solução: Ocorre sempre que um caminho é executado no programa em teste, independentemente dele ser o pretendido ou não. Ou seja, a cada avaliação de uma solução candidata do Algoritmo Genético a tupla [caminho, dado de entrada respectivo] é armazenada na base de soluções, alimentando-a com novos casos potencialmente úteis à execução de caminhos similares posteriormente.

A adequação deste paradigma à geração de dados de teste deve-se à alta complexidade deste problema. Encontrar valores de entrada para o programa que satisfaçam as restrições representadas pelos predicados existentes ao longo do caminho pretendido é uma tarefa complexa e dispendiosa do ponto de vista computacional. Neste caso, recuperar dados de entrada executados anteriormente que satisfaçam as restrições referidas pode significar uma substancial economia em termos de computação e tempo dispendidos. Quando um dado de entrada que execute o caminho pretendido não é encontrado na base de soluções é feita a recuperação baseada na *métrica de similaridade*. Isto permite que sejam utilizados dados de entrada com a propriedade de satisfazer um conjunto de restrições necessárias para que o caminho pretendido seja executado para compor a população inicial do algoritmo genético.

Reuso de Soluções Passadas em Programas Modificados

No processo de teste defeitos podem ser revelados pela observação de falhas do software na execução dos dados de teste. Nesses casos, torna-se necessário realizar a depuração do programa; isto é, a identificação e a correção desses defeitos (Capítulo 2). Tem-se então a realização do teste de regressão, feito no programa modificado para estabelecer a confiança de que as mudanças realizadas foram corretas e que não afetaram as partes não alteradas do programa [RH96] . Quando um programa em teste é modificado para corrigir algum defeito revelado tem-se, possivelmente, uma alteração semântica do mesmo. Não é possível portanto garantir que os dados de entrada contidos na base de soluções continuarão executando os respectivos caminhos. Ou seja, os casos contidos em uma base de soluções podem tornar-se inconsistentes quando são feitas modificações no programa a ela associado. Tendo em mente este problema podem ser estabelecidas as seguintes estratégias de tratamento:

[i]*Remoção da base de soluções:* A cada modificação realizada no programa deve-se remover a base de soluções. Deste modo, a nova base formada para o programa modificado terá a sua consistência garantida enquanto este não for novamente modificado. Esta opção, no entanto, implica na possibilidade de deixar de recuperar dados de entrada potencialmente úteis na geração de dados para o programa modificado. Isto pode ocorrer se os caminhos pretendidos não contiverem o nó do grafo no qual foi feita a manutenção do programa; neste caso, os caminhos que seriam recuperados da base de soluções não seriam afetados pela manutenção e, portanto, a

consistência dos casos recuperados seria garantida. Remover a base nestas situações significa deixar de recuperar estes dados de entrada, com o conseqüente acréscimo no custo da busca;

[ii] *Recuperação de dados associados a caminhos com similaridade limitada:* Considerando que a modificação foi realizada em um determinado nó n do grafo do programa, este tratamento consiste em limitar a recuperação de dados de entrada associados aos caminhos com um limite superior de similaridade L. O valor de L corresponte à posição da primeira ocorrência do nó n no caminho pretendido. Exemplificando: se o nó sete do grafo sofreu uma manutenção e o caminho pretendido for: 1 2 4 5 7 9 11 13, o nível máximo de similaridade para a recuperação (valor L) será cinco (posição do nó sete no caminho pretendido). Garante-se desta forma a consistência dos dados recuperados caso o caminho pretendido contenha o nó do grafo onde foi feita a modificação do programa;

[iii] *Verificação dinâmica da consistência do caso recuperado:* Ao se recuperar dados de entrada associados aos caminhos existentes na base de soluções é possível, considerando o caminho pretendido, verificar se o caso recuperado é ou não consistente; ou seja, se ele foi ou não afetado pela modificação realizada no programa. Isto é possível confrontando-se dois valores da métrica de similaridade. O primeiro valor é a métrica de similiaridade computada do caminho recuperado da base em relação ao caminho pretendido. O segundo valor é a métrica de similaridade computada do caminho efetivamente executado no programa modificado, pelo dado de entrada recuperado da base, em relação caminho pretendido. Se estes valores forem diferentes entre si, então conclui-se que o dado de entrada existente na base foi afetado pela manutenção do programa e, portanto, a sua inclusão na população inicial do Algoritmo Genético deve ser reavaliada.

Este processo de reavaliação da similaridade pode permitir que sejam incluídos na população inicial do Algoritmo Genético apenas dados de entrada aparentemente não afetados pela modificação do programa [1], ou os afetados que executem caminhos com um alto nível de similaridade [2]. Deste modo podem ser incluídos na população inicial mesmo os dados de entrada que executam caminhos que passam pelo nó do grafo onde houve a modificação, mas não são afetados por ela. Por outro lado, está associado a este tratamento um considerável acréscimo de custo computacional, relativo à reexecução dos dados de entrada existentes na base de soluções que sejam candidatos à recuperação, segundo à medida de similaridade verificada a partir do caminho existente na base.

[1] Ou seja, dados de entrada que executem no programa modificado o mesmo caminho que executavam no programa original.

[2] Isto é, dados de entrada que, apesar de executarem no programa modificado caminhos diferentes do que executavam no programa original, mantenham um alto nível de similaridade com o caminho pretendido.

Está sendo utilizada atualmente a estratégia *i*; entretanto, as estratégias *ii* e *iii* podem ser consideradas em desenvolvimentos futuros visando aprimorar a utilização da base de soluções na geração de dados de teste para programas modificados.

4.6.4 Adaptação de Soluções Utilizando Algoritmo Genético

Merece destaque o papel desempenhado pelo Algoritmo Genético no que tange à utilização do Raciocínio Baseado em Casos. A utilização deste algoritmo permite a automação do reparo e adaptação de soluções. Isto significa que mesmo se nenhum dado de entrada que execute o caminho pretendido for encontrado na base, não será necessária intervençao humana no processo de geração de dados — exceto no tratamento de elementos não executáveis, questão abordada na Seção 4.7.

Na abordagem elaborada a adaptação das soluções é feita de forma automática pela composição de soluções pertinentes. Os conjuntos de valores de entrada para o programa são recuperados da base de soluções considerando a métrica de similaridade de caminhos. As soluções recuperadas formam a população inicial do algoritmo genético que é o responsável pelo processo de adaptação. Este algoritmo manipula as soluções candidatas presentes na população inicial através da aplicação dos operadores genéticos: seleção, recombinação e mutação. Em um processo iterativo é realizada a otimização das soluções presentes na população do Algoritmo Genético de forma que elas se tornem cada vez mais adaptadas ao novo objetivo, isto é, à execução do novo caminho pretendido.

Tem-se, portanto, uma adaptação próxima da *Substituição baseada em casos*, com a composição das soluções sendo realizada através dos operadores genéticos. A aplicação do operador de recombinação substitui valores das variáveis de entrada utilizados em cada solução tomando-as duas a duas, o operador mutação introduz alterações aleatórias nestes valores, o operador seleção distingüe as boas soluções das ruins direcionando a busca. Esta distinção — feita pela função de ajuste — é análoga à métrica de similiaridade utilizada para a recuperação dos casos da base de soluções.

A utilização conjunta do paradigma Raciocínio Baseado em Casos e dos Algoritmos Genéticos é favorecida pela existência de aspectos conceituais comuns às duas abordagens. Como destacado na Seção 4.2.2 a busca utilizando os Algoritmos Genéticos dá-se em um processo no qual diversos esquemas competem entre si de uma maneira tal que os esquemas que levam às boas soluções tendem a ter uma maior participação em gerações futuras. Esta dinâmica representa um sistema de memória primitiva em que ocorre de forma simultânea a exploração do domínio da busca e a utilização do conhecimento adquirido para direcioná-la. A utilização do Raciocínio Baseado em Casos viabiliza a recuperação e uso de informações advindas de explorações realizadas no domínio da busca em ocasiões anteriores. Isto é, o conhecimento adquirido sobre o programa durante a busca de dados que executem outros caminhos tornam-se disponíveis e são utilizados pelo Algoritmo

Genético. Deste modo, o Raciocínio Baseado em Casos propicia uma "memória de longo prazo" útil para a aplicação do Algoritmo Genético. Sob a ótica deste algoritmo a recuperação de indivíduos da base de soluções representa a identificação e recuperação de características genéticas existentes no passado, úteis para a solução de um novo problema de adaptação.

4.7 Questão da Não Executabilidade no Teste Estrutural

Uma questão complexa no teste estrutural de software diz respeito à existência de *caminhos não executáveis*, caminhos para os quais não existe algum conjunto de valores de entrada do programa que os executem. Determinar caminhos não executáveis é uma questão indecidível. Segundo Clarke este problema é análogo ao problema da parada ("halting problem") e, portanto, não solucionável [Cla76].

Segundo [HH85] um caminho não executável é um caminho que nunca pode ser executado porque isto requereria que predicados contraditórios fossem satisfeitos. Segundo este autor, um grande número de caminhos é não executável, o que faz com que a tarefa da geração de dados seja extremamente difícil.

O efeito da não executabilidade na geração de dados de teste está relacionado a um requisito para o encerramento desta atividade. Segundo [MYE79] a satisfação de um critério é um item a ser atingido para o encerramento do teste. Elementos requeridos pelos critérios podem ser não executáveis devido à inexistência de caminhos executáveis que os cubram. Neste caso, para a satisfação destes critérios é necessário identificar tais elementos, uma tarefa complexa e de difícil automação.

Além disso, a existência de caminhos não executáveis altera de forma significativa propriedades teóricas como a *relação de inclusão* no casos dos critérios definidos por Rapps e Weyuker [RW85, FW88]. Segundo Maldonado a relação de inclusão dos critérios Potenciais Usos em relação a todos os ramos permanece inalterada para esses critérios mesmo na presença de caminhos não executáveis [Mal91].

Vergilio [Ver92] destaca as principais abordagens presentes na literatura relacionadas a este tema:

determinação: destaca-se o trabalho de Frankl [FW86, Fra87] que propôs heurísticas para determinação de elementos não executáveis. São utilizadas técnicas de execução simbólica e de análise de fluxo de dados para determinar associações não executáveis;

caracterização: visa identificar as principais causas de não executabilidade [HH85, Ver92];

previsão: destacam-se os trabalhos de Malevris e outros [MYV90] e Vergilio e outros [Ver92] que estudam a influência do número de predicados na executabilidade de um caminho. Segundo estes trabalhos quanto maior o número de predicados de um caminho maior a probabilidade do caminho ser não executável. Malevris mostra através de um experimento considerando 648 caminhos que a probabilidade de um caminho ser executável decai exponencialmente com o número de predicados atravessados.

4.7.1 Seleção de Caminhos para Satisfação de Critérios de Teste Estrutural

Em relação à geração de dados o tratamento desta questão dá-se na seleção de caminhos no programa para exercitar os elementos requeridos pelos critérios e também no processo de geração dos dados que os executem. No primeiro caso um possível tratamento é escolher caminhos com menor chance de serem não executáveis [MYV90].

Em [FB97] é abordado o problema da seleção de um caminho no programa da entrada até um determinado ponto. É destacado que, considerando o objetivo estabelecido, deve-se minimizar o número de predicados que influenciam o ponto e não do caminho como um todo. É proposta então uma abordagem para selecionar caminhos factíveis buscando minimizar este número. É aplicada a técnica denominada *slice principal* que determina apenas os comandos que podem afetar um conjunto de variáveis referenciadas em um determinado ponto do programa. Isto faz com que sejam reduzidos o número de predicados considerados durante a fase de geração de dados de entrada.

Em [BGS97] é definida uma técnica para identificar alguns caminhos não executáveis através de *análise estática de correlação de ramos*. Essencialmente esta correlação ocorre para um ramo, em relação a um caminho, se a sua saída (ou avaliação) pode ser determinada pelas saídas de ramos ou comandos anteriores, em tempo de compilação. É também apresentada uma técnica para análise de associações definição-uso não executáveis.

Em [Per97] são analisadas estratégias de seleção de caminhos para satisfação de critérios de teste estrutural, tema relacionado aos trabalhos descritos anteriormente.

Em [BJVM97] tem-se a descrição de módulos de tratamento de elementos não executáveis integrados à POKE-TOOL. Além da heurística de identificação de associações não executáveis [Ver92], tem-se a descrição do módulo *pokepadrao*. Este módulo tem a função de automatizar o tratamento de *padrões de não executabilidade* identificados pelo testador. A partir da identificação de componentes estruturais não executáveis (nós, arcos, associações ou subcaminhos) são removidos da lista de requisitos de teste os elementos não executáveis devido ao padrão.

Estes trabalhos enfocam a identificação a priori de elementos não executáveis. En-

tretanto, conforme destacado em [BGS97] entre outros autores, a solução geral deste problema é impossível. Portanto, mesmo com a aplicação dessas técnicas, elementos não executáveis podem não ser identificados. Nestes casos o problema é transferido para a etapa de geração de dados de teste.

4.7.2 Não Executabilidade e Geração de Dados de Teste

A partir dos caminhos selecionados o processo de geração de dados deve maximizar a eficácia na identificação dos caminhos não executáveis possivelmente selecionados, assim como minimizar o custo dispendido na busca caso algum caminho seja não executável sem que esta identificação tenha sido possível.

Nesta linha destacam-se alguns trabalhos:

- Em [RSBC76] tem-se uma abordagem baseada em processamento simbólico (expressões simbólicas denotam a evolução das variáveis no programa). São detectadas cláusulas simples contraditórias nos predicados, possibilitando a identificação de alguns dos caminhos não executáveis. Devido às características do algoritmo de busca (chamada de "tentativa e erro sistemática") em caso de falha na deteção é gerado um grande número de operações "backtracking". É destacada a necessidade de uma abordagem que permita a identificação antecipada da não executabilidade.

- Em [Cla76] trambém é utilizado o processamento simbólico. Sempre que uma transferência condicional é encontrada no programa restrições representando o ramo respectivo são geradas. Estas restrições são escritas em arquivo e posteriormente simplificadas e resolvidas. Esta abordagem permite a identificação da não executabilidade de caminhos cujos predicados formem uma função linear das variáveis de entrada. Também é afirmado que idealmente as restrições deveriam ter a sua consistência checada logo que geradas, a fim de evitar processamento simbólico desnecessário de todo o caminho. Esta solução não é adotada por problemas de armazenamento segundo a autora.

- [BEK75] apresenta um tratamento semelhante aos anteriores.

- [GBR98] transforma o programa estaticamente em um sistema de restrições. São aplicadas técnicas de solução de restrições para verificar a existência de algum caminho executável que atinja o ponto desejado do programa e para gerar dados de teste. Restrições geradas são utilizadas em um passo preliminar para detectar alguns dos caminhos não executáveis.

- Em [FK96] é apresentada a *"chaining approach"* para a geração de dados de teste. Esta técnica é uma extensão de [Kor90] e incorpora a seleção de caminhos no processo de geração de dados. São utilizadas informações sobre dependência de dados para

determinar comandos do programa que afetam a execução de outros. Deste modo, são definidas "seqüências de eventos" (seqüências de nós) que devem ser executadas antes de atingir um dado predicado, de forma que o algoritmo de busca tenha maior chance de satisfazê-lo. Quando o processo de busca falha na tentativa de encontrar dados de entrada que satisfaçam um determinado predicado uma nova tentativa é feita alterando os nós a serem executados antes deste predicado.

- Segundo Gupta e outros [GMS98] um dos maiores desafios das abordagens baseadas em execução é o impacto dos caminhos não executáveis. Este trabalho utiliza um método denominado "iterative relaxation", utilizado em análise numérica para aprimorar soluções aproximadas de equações representando raízes de uma função. A técnica utilizada garante que, para condições de caminho que sejam representadas como uma função aritmética linear dos dados de entrada, é possível garantir a identificação da não executabilidade quando ela existir. Deve-se notar que tal identificação só é possível se todos os predicados do caminho representarem funções lineares das variáveis de entrada. Caso algum predicado represente uma função não linear a técnica não é aplicável.

Os trabalhos relacionados à geração de dados de teste analisados abordam a não executabilidade destacando a indecidibilidade desta questão. O enfoque comum é restringir o domínio de caminhos a uma classe, garantindo a identificação de não executabilidade para caminhos pertencentes a esta classe. Apresenta-se a seguir um tratamento de enfoque distinto para a questão.

4.7.3 Tratamento Dinâmico Para a Questão da Não Executabilidade de Caminhos: Identificação de Potencial Não Executabilidade

A abordagem aqui apresentada difere das anteriores citadas. Não é feita a aplicação de execução simbólica e prova de teoremas, também não são aplicadas técnicas de solução de funções lineares. Ao contrário, é feito um monitoramento do progresso da busca realizada pelo Algoritmo Genético.

Como destacado na Seção 4.2.1 a busca com Algoritmos Genéticos é realizada com uma população de soluções que evoluem seguindo regras probabilísticas. A evolução da busca pode ser avaliada pelo monitoramento de medidas de performance. Esta abordagem é utilizada por De Jong no contexto da otimização de funções utilizando Algoritmos Genéticos [Gol89]. A idéia deste trabalho é a de avaliar a performance da busca computando medidas baseadas nos valores da *função de ajuste* utilizada. O objetivo foi o de quantificar a performance do Algoritmo Genético considerando diferentes problemas e operadores genéticos. No contexto da geração de dados de teste este monitoramento é útil

para a identificação de situações de *potencial não executabilidade* do caminho pretendido, associadas à baixa performance persistente da busca.

O valor *média de ajuste da população* reflete a média de qualidade das soluções da população e pode ser utilizado para aferir de forma bastante confiável o progresso da busca. Na geração de dados para caminhos executáveis, observa-se um progresso contínuo da média de ajuste, na medida em que os predicados vão sendo alcançados e solucionados pelo Algoritmo Genético. Por outro lado, a tentativa de gerar dados para caminhos não executáveis resulta, invariavelmente, em uma ausência persistente de progresso deste valor; isto ocorre porque, neste caso, algum predicado do caminho é não factível, o que impossibilita o progresso da busca.

A identificação de ausência de progresso é feita através da seguinte heurística:

Se

i) $aj[i] - aj[i-1] \leq \delta \; \forall \; i, \; j \leq i \leq (j + NL)$ e

ii) $aj[j + NL] - aj[j] \leq \Delta$

então: Situação de *potencial não executabilidade* identificada;

sendo:

$aj[i]$: ajuste médio da população na geração i;

$aj[i-1]$: ajuste médio da população na geração $i-1$, imediatamente anterior à i;

j: geração na qual a relação [i] foi inicialmente observada;

δ: Valor limite abaixo do qual se considera situação de ausência de progresso para duas gerações sucessivas;

Δ: Valor limite abaixo do qual se considera situação de ausência de progresso acumulado por NL gerações;

NL: Número de gerações requerido para a satisfação da condição [i];

Esta heurística é implementada através do monitoramento do valor do ajuste médio da população de soluções. A cada nova geração este valor é apurado e comparado com o da geração anterior. Caso o aumento do valor do ajuste médio for inferior a δ, tem-se o incremento de um contador. Se este contador atingir NL tem-se a aplicação da regra [ii]. Em qualquer situação se o aumento do ajuste médio for superior a δ tem-se o re-início da contagem.

A regra [ii] é aplicada sempre que a busca apresentar progresso sucessivo inferior a δ durante um número de gerações igual a NL. É então verificado se desde o início da contagem (geração j) até a geração corrente ($j + NL$) houve um progresso inferior à Δ. Caso positivo, tem-se identificada a situação de potencial não executabiliade do caminho pretendido.

O valor NL determina o quão persistente necessita ser o não progresso da busca para que o testador seja advertido. Valores menores para NL permitem uma identificação mais rápida dos caminhos não executáveis, mas com uma baixa confiabilidade. Valores maiores, por outro lado, fazem com que esta identificação ocorra apenas após grandes gastos de recurso computacional. Neste caso, entretanto, a chance de erro de avaliação (identificar erroneamente um caminho executável como não executável) tende a ser menor.

Definiu-se a seguinte equação para o cálculo de NL:

$$NL = K1 + K2 \times (Np + Nv)$$

sendo:

NL: Número de gerações requerido para a satisfação da condição [i];

$K1$: Valor constante;

Np: Número de predicados do caminho pretendido;

Nv: Número de variáveis de entrada do programa (número de parâmetros + número de variáveis recebidas pelo teclado + número de variáveis globais);

$K2$: Fator de controle da influência de Np e Nv em NL;

Esta equação modela a complexidade do problema da geração de dados como sendo função do número de nós do caminho e do número de variáveis de entrada do programa. Isto faz com que para problemas mais complexos (maiores números de nós do caminho e de variáveis de entrada) a heurística seja adaptada, requerendo um número maior de gerações sem progresso na busca para que esta situação seja informada ao testador. Esta adaptação visa ajustar de forma automática a heurística ao problema particular (caminho pretendido), para que um caminho não executável possa ser distingüido de um caminho "difícil" de ser executado.

Uma relação exponencial entre o comprimento do caminho e o custo da busca foi obtida empiricamente em [GN97]. Foi verificado também um relacionamento linear entre o número de variáveis de entrada e este custo.

Deve-se notar que a potencial não executabilidade é identificada se e somente se:

A diferença no valor do ajuste médio for inferior a δ durante NL gerações sucessivas; e

A diferença no valor do ajuste médio for inferior a Δ considerando a geração inicial da aplicação da heurística (geração j) até a geração corrente $(j + NL)$;

Atualmente utilizam-se os seguintes parâmetros: $\delta = 0,50$; $\Delta = 0,30$; $K1 = 30$ e $K2 = 3$. Caso sejam satisfeitas as regras i e ii a busca é provisoriamente interrompida e tem-se a identificação da potencial não executabilidade do caminho pretendido. Neste caso o testador é informado do predicado suspeito de provocar a não executabilidade do caminho e dos valores: δ, Δ e NL. É emitida uma mensagem do tipo:

```
DataGen - Identificada situacao de possivel nao executabilidade
do caminho pretendido.

Numero da Geracao: 146
Numero da Tentativa: 14600

O ajuste medio da populacao teve progresso acumulado inferior a 0.30
e progresso sucessivo inferior a 0.50 nas ultimas 114.000000 geracoes.

Progresso acumulado em 114.000000 geracoes: 0.025000

Caminho Pretendido:  1  2  3  5  6  14  16  18  19  20  21  2  22

Numero de nos corretamente executados da melhor solucao: 3
Ultimo no corretamente executado: 3
Verifique a executabilidade do arco ( 3 , 5 ) em relacao aos predicados
anteriores do caminho.

Interrompe a busca ? (s|n)
```

Neste caso, ocorrerá que a maioria das soluções candidatas da população executará o caminho pretendido até o predicado que provoca a não executabilidade por não ser factível. Este predicado, suspeito de provocar a não executabilidade, é identificado. Esta identificação é feita pela análise do caminho executado no programa pela melhor solução candidata, pois esta certamente atingiu o predicado em questão.

4.7.4 Enfoque Dinâmico × Estático Para o Tratamento da Não Executabilidade

No caso de abordagens que utilizam análise estática simbólica e provadores de teorema a identificação de não executabilidade é determinística, isto é, dado que os algoritmos identificam o elemento requerido (ou caminho no programa) como sendo não executável pode-se com certeza eliminar este elemento do conjunto de requeridos pelo critério (ou

selecionar um novo caminho no programa). Entretanto, apenas uma classe restrita de caminhos é passível deste tratamento.

Naturalmente a solução exata do problema, que identifique todos elementos requeridos não executáveis, não é factível dada a indecidibilidade do problema.

Por outro lado, o enfoque dinâmico aqui apresentado é aplicável indistintamente, independentemente do tamanho do programa, de tipos de variáveis envolvidas e de pré-requisitos sobre equações de caminhos tratáveis. No entanto, não se pode afirmar que trata-se de uma heurística de identificação de não executabilidade, visto que sempre existirá a possibilidade de avaliação incorreta da (não)executabilidade de caminhos.

À medida que os recursos dispendidos na busca crescem sem que os valores de entrada que executem o caminho sejam encontrados, aumenta o nível de confiança na não executabilidade deste caminho. Entretanto, mesmo após um número de iterações expressivamente elevado, não é possível classificar o caminho como não executável de forma confiável. Isto porque, na técnica dinâmica utilizada, sempre existirá a dúvida de se o conjunto de valores de entrada adequado não foi encontrado por deficiência da busca, ou se este conjunto não existe: situação na qual o caminho é não executável.

A utilidade da informação gerada neste tipo de tratamento relaciona-se aos seguintes pontos:

A informação de qual predicado do caminho pretendido é o suspeito de causar a não executabilidade pode facilitar o trabalho do testador. Neste caso pode-se analisar o código do programa para a confirmar ou não a executabilidade deste predicado, focalizando a análise nos fluxos de dados e de controle que o influenciam; pode-se verificar, por exemplo, se o predicado suspeito contradiz os demais anteriores do caminho originando a não executabilidade. Esta tarefa é comumente mais simples do que analisar o caminho como um todo, devido à redução do escopo de análise;

Uma outra opção consiste em tomar como verdadeira a identificação feita pela ferramenta sem maiores análises. Esta opção implica no risco de considerar como não executável um caminho que não o é. Isto pode significar deixar de exercitar um elemento requerido pelo critério por considerá-lo, erroneamente, não executável. Esta postura contraria o requisito para que se considere encerrada a atividade de teste: satisfazer o critério de teste selecionado [MYE79].

Entretanto, sob um ponto de vista prático esta solução não deve ser descartada a priori. Deixar de testar elementos requeridos no caso da utilização de critérios de teste exigentes pode ser aceitável, desde que a chance de erro na avaliação da não executabilidade seja minimizada o quanto possível e que o software em teste não apresente altos requisitos de confiabilidade. Caso estas restrições sejam satisfeitas é possível adotar este tratamento.

Neste caso a questão da geração de dados de teste seria completamente automatizada, pois não haveria intervenção do testador no tratamento da não executabilidade. Naturalmente, continua sendo necessária esta intervenção para a avaliação de se o dado de entrada provocou ou não a revelação de defeito, ou seja, se a saída obtida corresponde a esperada tendo em vista a especificação do programa.

Merece destaque o fato de que no caso da aplicação de técnicas de tratamento de não executabilidade baseadas em análise estática, identificam-se como não executáveis apenas um sub-conjunto dos elementos que de fato o são. Os elementos restantes, que "escapam" do tratamento estático, podem também ser não executáveis e cabe ao testador identifica-los. Esta é uma tarefa difícil, tediosa e, por ser baseada em atividade humana, é sujeita a erros. Portanto, torna-se necessário confrontar o índice de erro humano neste caso com o da heurística de avaliação dinâmica da não executabilidade.

Existe uma analogia entre este tratamento para a questão da não executabilidade e o proposto em [BEK75, Cla76]. Nestes trabalhos — baseados na idéia de processamento simbólico — as restrições associadas aos predicados são analisadas pelo resolvedor de inequações na medida em que são geradas no processamento simbólico ao longo do caminho. Isto permite a descoberta antecipada de restrições inconsistentes com as anteriores do caminho. O tratamento proposto pode ser visto como uma adaptação desta abordagem para a geração dinâmica; entretanto, ao invés de tentar provar a não executabilidade buscando explicitamente contradições nos predicados, isto é feito através da apuração e análise de parâmetros que refletem a dinâmica da busca.

Capítulo 5

Aplicação da Ferramenta de Geração Automática de Dados de Teste

Este capítulo descreve o experimento conduzido para a validação da ferramenta de geração automática de dados de teste. São relatados os resultados obtidos as análises feitas. São também identificados aspectos que podem ser aprimorados a partir das observações realizadas.

Conforme destacado no Capítulo 3 têm-se duas etapas na abordagem para a geração de dados de teste utilizada:

[i] Seleção de um caminho completo no programa que cubra cada elemento requerido pelo critério de teste; e

[ii] Geração de um dado de entrada que execute o caminho selecionado.

Este trabalho abordou o problema descrito no Item [ii]. A estratégia de validação, portanto, deve avaliar o desempenho da ferramenta na geração de valores de entrada que executem um determinado caminho do programa. Adicionalmente, pretende-se verificar a pertinência das abordagens propostas. Essencialmente será avaliada a influência do Reuso de Soluções Passadas, modelado utilizando o paradigma Raciocínio Baseado em Casos, no custo da geração de dados. Será também verificada a eficácia da Heurística de Identificação Dinâmica de Potencial Não Executabilidade.

A comparação direta de resultados com outras propostas é difícil por causa das diferenças de linguagem tratada (Pascal, Fortran, Ada, C e sub-conjuntos destas) e do objetivo definido para o teste (exercitar um nó do grafo, uma sequência de nós ou um caminho completo do programa). Devido a este fato a validação será feita através da comparação da eficácia da geração de dados utilizando a ferramenta proposta e utilizando uma busca aleatória, implementada em um módulo integrado à mesma. A comparação

de ferramentas desenvolvidas com a geração aleatória é utilizada em alguns trabalhos analisados [JSE96, MMSC97, FK96]. No contexto deste trabalho, a diferença entre o custo da geração aleatória e o custo utilizando a ferramenta pode ser vista como uma estimativa de ganho em relação ao esforço computacional para gerar dados que executem um determinado caminho do programa, associado a algum requisito de teste.

5.1 Experimento de Validação

Tendo em vista este objetivo da ferramenta e visando a validação das soluções elaboradas, foi idealizado um experimento que permite a avaliação da eficácia da geração de dados em relação aos seguintes aspectos:

- Programas envolvendo diferentes tipos de variáveis de entrada;
- Caminhos cujos predicados sejam expressões utilizando os diversos tipos de operadores relacionais considerados, incluindo também predicados compostos com operadores lógicos;
- Caminhos cujos predicados envolvam diferentes tipos de variáveis;
- Caminhos que contenham diversas iterações de laços e que combinem estruturas de seleção e repetição.

Para tanto foram selecionados quatro programas que, no conjunto, permitem a avaliação dos aspectos salientados acima. Os programas 1 e 2 foram implementados para a avaliação do tratamento de variáveis tipo "float" e dos tipos "char" e "string", respectivamente. O programa 3, semelhante ao utilizado em [GN97], manipula variáveis do tipo inteiro e contém laços. O programa 4 é uma versão *C* para o *find* utilizado em [Fra87]. Estes programas são descritos nas seções seguintes. Todos foram instrumentados manualmente seguindo o Modelo de Instrumentação definido na Seção 4.5.

Para cada programa foi realizado o seguinte procedimento:

1. Foram selecionados caminhos completos executáveis para formar o conjunto de caminhos pretendidos;
2. Foram gerados dados de entrada que executem cada caminho deste conjunto. Os caminhos foram tomados um por vez, do primeiro ao último selecionado. Isto representa uma simulação de utilização da ferramenta para gerar dados associados a diversos caminhos a serem executados no programa em teste;

   A geração de dados foi feita de três modos distintos:

- Modo [**i**]: Utilizando o Algoritmo Genético para a busca dos dados de entrada e recuperando dados presentes na base de soluções para compor a população inicial do Algoritmo Genético;
- Modo [**ii**]: Utilizando o Algoritmo Genético para a busca dos dados de entrada mas sem recuperar dados da base de soluções. Nestes casos a população inicial foi toda gerada aleatóriamente;
- Modo [**iii**]: Simulando uma busca aleatória pelos dados de entrada;

3. Para cada programa foi registrada a descrição das variáveis de entrada (interface informada à ferramenta);

4. Para cada programa e modo de execução foram registrados os parâmetros utilizados no Algoritmo Genético e parâmetros sobre o reuso de soluções da base;

5. Os passos 2.i e 2.ii foram repetidos dez vezes e o passo 2.iii cinco vezes para cada conjunto de caminhos selecionado. O objetivo é reduzir o efeito de variações de caráter aleatório nos resultados. No caso do passo 2.i após cada repetição (isto é, a geração de dados para todos os caminhos selecionados) o conteúdo da base de soluções era eliminado — de outro modo nas tentativas subseqüentes a solução seria encontrada sempre nesta base;

6. Em cada tentativa de geração de dados para executar o caminho pretendido foi apurado o número de avaliações de indivíduo necessárias para que a solução fosse obtida, isto é, para que o dado de entrada que executa o caminho fosse descoberto. Este valor é o número de execuções do programa e reflete o custo da busca para o caminho em questão e o modo de execução considerado. É referido por NE.

7. No caso do modo de execução 2.i foi registrado sempre quando o dado de entrada que executa o caminho foi encontrado na base de soluções, dispensando a realização da busca com o Algoritmo Genético. Nestes casos tem-se o símbolo ^{EB} nos campos respectivos das tabelas de resultados;

8. Nos modos de execução 2.i e 2.ii foi registrado sempre quando foi identificada a situação de potencial não executabilidade do caminho pretendido. Nestes casos tem-se o símbolo ^{Pne} nos campos respectivos das tabelas de resultados;

9. Para cada caminho pretendido e modo de execução (**i**, **ii**, ou **iii**) foi calculada a média aritmética do número de execuções necessárias para que um dado de entrada que executa o caminho fosse descoberto, considerando as várias tentativas (média dos valores NE respectivos). Valor referido por MEC;

10. Para cada programa e cada modo de execução (**i**, **ii**, ou **iii**), foi calculada a média aritmética do número de execuções para todos os caminhos, considerando os valores médios do número de execuções de cada caminho (MEC). Valor referido por MEP;

Em relação ao procedimento anterior cabem algumas observações:

Foi apurado o tempo médio para a avaliação de 100 indivíduos considerando os vários programas. Este valor é utilizado para a estimação do tempo médio de busca dispendido em caminhos do programa, considerando cada modo de execução;

Para cada caminho pretendido, independentemente do modo de execução, foi limitado o valor máximo para o número de execuções do programa: $NE = 45.000$; para o programa 4 estabeleceu-se $NE = 100.000$ devido à maior complexidade observada para a geração de dados. Ao atingir este valor (correspondente ao limite 1125 gerações — ou 1000 para o programa 4 — com tamanho da população = 100) a busca é encerrada e constatada a falha na geração de dados. Em termos de tempo de execução tal valor para NE equivale a aproximadamente 2 horas e 7 minutos de busca (4 horas e 42 minutos para $NE = 100.000$). Nestes casos tem-se o símbolo ^{Fl} nos campos respectivos das tabelas de resultados e a utilização do valor limite (45.000 ou 100.000) para o cálculo de MEC;

Foi utilizada uma estação de trabalho Sun Ultra-1; 256Mb RAM; clock 143 MHz; sistema operacional SunOS 5.5.1; sistema de janelas OpenWindows 3.5.1. Nenhum controle sobre a prioridade do processo foi realizado. Do mesmo modo, não foram feitas restrições de acesso de outros usuários à máquina utilizada.

A busca aleatória, utilizada no item 2.iii do procedimento acima, foi simulada através de uma funcionalidade implementada e descrita a seguir:

Partindo-se da descrição das variáveis de entrada aceitas pelo programa é calculado o "offset" do indivíduo do Algoritmo Genético. Esta estrutura de dados é utilizada para representar cada dado de entrada gerado. Tem-se então um ciclo onde as soluções são aleatoriamente geradas e avaliadas até que um dado de entrada que executa o caminho seja encontrado, ou que o número limite de tentativas seja atingido ($NE = 45.000$ ou $NE = 100.000$). Em ambos os casos o número de tentativas é registrado. Cada solução candidata é gerada de forma independente e aleatória (utilizando o comando *random* da linguagem C). É renovada a "semente aleatória" (gerada pelo comando *time*, que retorna o tempo corrente em segundos) a cada nova tentativa.

5.1.1 Valores Computados

Conforme destacado nos passos 9 e 10 do procedimento de validação descrito anteriormente, para cada programa e cada modo de execução **i**, **ii** e **iii**, são computados os valores:

MEC: Média do número de execuções necessárias para que um dado de entrada que executa o caminho seja descoberto, valor computado para cada caminho pretendido do programa;

MEP: Média dos valores MEC para o programa.

Considerando cada programa e os valores MEP para os diversos modos de execução, são computados os seguintes *indicadores de eficiência*:

1) $\frac{MEPiii}{MEPi}$: Razão entre o número de execuções utilizando geração aleatória e utilizando Algoritmo Genético e reuso de soluções. Mostra a redução do custo computacional propiciado pela utilização da ferramenta em relação à geração aleatória para o programa considerado;

2) $\frac{MEPiii}{MEPii}$: Razão entre o número de execuções utilizando geração aleatória e utilizando Algoritmo Genético (sem o reuso de soluções). Permite avaliar o desempenho do Algoritmo Genético em relação à geração aleatória para o programa considerado;

3) $\frac{MEPii}{MEPi}$: Razão entre o número de execuções utilizando apenas o Algoritmo Genético e utilizando Algoritmo Genético e reuso de soluções. Permite avaliar a influência do reuso de soluções no custo da geração de dados para o programa em questão.

5.1.2 Parâmetros Utilizados

Para os programas 1 e 2 foram utilizados os parâmetros para o Algoritmo Genético:

- Probabilidade de Recombinação $(Pr) = 0,8$;
- Probabilidade de Mutação $(Pm) = 0,02$;
- Tamanho da População $(Tp) = 40$;

Para o programa 3 o Tamanho da População foi de 60 e os demais parâmetros foram os anteriores.

Para o programa 4 foi utilizado Tamanho da População de 100 e $Pm = 0,04$. O valor de Pr foi mantido.

O ajuste dos parâmetros do Algoritmo Genético foi baseado em valores típicos na literatura (descritos na Seção 4.4.4) e na breve observação do desempenho do Algoritmo Genético. Em geral, para problemas mais complexos (programas com um maior número de variáveis de entrada e com caminhos pretendidos possuindo um grande número de predicados), optou-se por uma maior diversidade de soluções (Tp maior) e uma maior exploração do domínio de busca (Pm maior), mesmo que em detrimento do desempenho (convergência mais lenta para a solução).

Em relação ao reuso de soluções passadas, no modo de execução **i** utilizou-se:

- Recuperação de soluções similares habilitada;
- Limite máximo de soluções recuperadas $= 0,5 \times Tp$;
- Nível mínimo de similaridade para recuperação de soluções $= 3$;
- Número máximo de casos inseridos na base para cada caminho pretendido $= 15.000$;

No modo de execução **ii** a recuperação foi desabilitada e no modo **iii** esta opção não se aplica.

Durante o experimento os parâmetros da heurística de identificação dinâmica de não executabilidade foram mantidos constantes nos seguintes valores: $\delta = 0,50$; $\Delta = 0,30$; $K1 = 30$ e $K2 = 3$.

5.2 Resultados Obtidos

Nesta seção são mostrados os dados apurados na aplicação do procedimento de validação definido anteriormente. São destacados os resultados para cada programa separadamente e são feitas análises iniciais. O código fonte e o grafo de fluxo de controle dos programas, assim como os caminhos selecionados para teste estão no Apêndice B.

As tabelas presentes nesta seção mostram os valores apurados de número de execuções necessárias para que as soluções fossem obtidas (NE).

5.2.1 Programa 1

Este programa recebe três variáveis de entrada tipo *float*, lidas do teclado. O programa computa valores baseados nestas variáveis e realiza comparações. Os predicados envolvem expressões utilizando variáveis do tipo float e diferentes operadores relacionais. São utilizados também predicados compostos com operadores lógicos.

Interface informada à ferramenta:

x, y, z: tipo "float"; ordem de grandeza $= 50$; precisão $= 2$.

A título de exemplo: a execução do caminho 4 requer que as seguintes restrições (expressas em função das variáveis de entrada x, y e z) sejam satisfeitas:

$$(z > y)\ and\ (y > x)\ and\ (z > x + y)\ and\ (0 \leq (x * y) - z \leq 5)$$

Indicadores de Eficiência:

Tabela 5.1: Programa1: Valores para NE utilizando Algoritmo Genético e Reuso de Soluções

Caminho / Tentativa	1	2	3	4	5
1	2	5	1^{EB}	648	1^{EB}
2	1	3	11	377	1^{EB}
3	3	1^{EB}	9	216	111
4	2	1^{EB}	2	424	1^{EB}
5	1	1^{EB}	9	257	105
6	1	2	3	212	67
7	2	1^{EB}	11	426	1^{EB}
8	9	1^{EB}	1^{EB}	383	209
9	1	7	1^{EB}	483	465
10	1	1	14	407	1^{EB}
Totais:	23	23	62	3833	962
MEC:	2,3	2,3	6,2	383,3	96,2
MEP:	98,1				

Tabela 5.2: Programa1: Valores para NE utilizando Algoritmo Genético

Caminho / Tentativa	1	2	3	4	5
1	1	3	16	491	212
2	1	4	4	884	18
3	3	1	2	408	61
4	1	1	6	69	99
5	1	3	27	639	33
6	3	4	23	400	75
7	2	2	4	50	255
8	3	1	4	463	90
9	2	1	3	474	35
10	1	1	10	53	142
Totais:	18	21	99	3931	1020
MEC:	1,8	2,1	9,9	393,1	102,0
MEP:	101,8				

Tabela 5.3: Programa1: Valores para NE utilizando Busca Aleatória

Caminho / Tentativa	1	2	3	4	5
1	2	11	9	5652	96
2	1	6	16	1032	138
3	1	2	2	5404	293
4	2	6	4	3618	91
5	3	10	23	5621	369
Totais:	9	35	54	21327	987
MEC:	1,8	7,0	10,8	4265,4	197,4
MEP:	896,48				

1) $\frac{MEPiii}{MEPi} = 9,14$;

2) $\frac{MEPiii}{MEPii} = 8,81$;

3) $\frac{MEPii}{MEPi} = 1,04$;

Os valores 1 e 2 mostram que o Algoritmo Genético requereu em média 9 vezes menos execuções do programa para a solução fosse encontrada. O valor 3 mostra que o reuso de soluções possiblitou uma redução de aproximadamente 4% no custo da busca.

Os caminhos 1, 2 e 3 estão associados a restrições de fácil satisfação. Nestes casos os três modos de execução tendem a ser equivalentes no valor de NE. Nos modos de execução **i** e **ii** em todas as tentativas a solução foi descoberta ainda na avaliação da população inicial do Algoritmo Genético, seja por geração aleatória ou por recuperação da base de soluções. Para estes caminhos variações nos valores de MEC nos diversos modos de execução podem ser vistas como aleatórias. O caminho 4, devido à maior complexidade, resultou em maiores valores para NE e possibilitou que o Algoritmo Genético se destacasse em relação à geração aleatória. Devido ao pequeno número de caminhos pretendidos e à relativa facilidade na geração de dados, a base de soluções manteve-se limitada (com poucos casos). Isto restringiu a influência do reuso de soluções no valor de MEP.

5.2.2 Programa 2

Este programa realiza a divisão inteira entre dois números e fornece o quociente e o resto. Os predicados envolvem comparações de variáveis do tipo inteiro computadas. A estrutura do programa contém laços com comandos de seleção interiores aos mesmos. Tem-se neste caso a validação da ferramenta para a geração de dados para programas que contenham tais estruturas. Verifica-se também a situação na qual predicados podem ser atravessados mais de uma vez na mesma execução (comandos de seleção interiores a laços).

Tabela 5.4: Programa2: Valores para NE utilizando Algoritmo Genético e Reuso de Soluções

Caminho / Tentativa	1	2	3	4	5	6	7	8	9	10
1	1	498	1^{EB}	1^{EB}	144	1^{EB}	1^{EB}	1^{EB}	1^{EB}	1^{EB}
2	1	435	1^{EB}	1^{EB}	88	1^{EB}	1^{EB}	224	1^{EB}	1^{EB}
3	35	599	1^{EB}	1^{EB}	329	1^{EB}	1^{EB}	442	1^{EB}	1^{EB}
4	3	299	1^{EB}	1^{EB}	545	1^{EB}	1^{EB}	96	1^{EB}	1^{EB}
5	10	338	1^{EB}	1^{EB}	314	1^{EB}	1^{EB}	428	1^{EB}	1^{EB}
6	5	519	1^{EB}	1^{EB}	80	1^{EB}	1^{EB}	143	1^{EB}	1^{EB}
7	4	576	1^{EB}	1^{EB}	39	1^{EB}	1^{EB}	305	1^{EB}	1^{EB}
8	4	552	1^{EB}	1^{EB}	234	1^{EB}	1^{EB}	730	1^{EB}	1^{EB}
9	3	805	1^{EB}	1^{EB}	346	1^{EB}	1^{EB}	1^{EB}	1^{EB}	1^{EB}
10	2	382	1^{EB}	1^{EB}	206	1^{EB}	1^{EB}	163	1^{EB}	1^{EB}
Totais:	68	5003	10	10	2325	10	10	2533	10	10
MEC:	6,8	500,3	1,0	1,0	232,5	1,0	1,0	253,3	1,0	1,0
MEP:	99,9									

Os caminhos selecionados incluem diversas iterações de laços, característica que tende a aumentar a dificuldade na geração de dados.

Interface informada à ferramenta:

$n1$, $n2$: tipo "int"; ordem de grandeza = 250.

Indicadores de Eficiência:

1) $\frac{MEPiii}{MEPi} = 69,16$;

2) $\frac{MEPiii}{MEPii} = 35,34$;

3) $\frac{MEPii}{MEPi} = 1,96$;

Os valores apurados revelam um número de execuções em torno de 69 vezes menor na utilização do Algoritmo Genético e reuso de soluções em relação à busca aleatória. Utilizando apenas o Algoritmo Genético o ganho foi de 35 vezes. O reuso de soluções permitiu que o Algoritmo Genético descobrisse os dados de entrada que executem os caminhos com aproximadamente a metade do número de execuções.

Resultados relativos aos caminhos 2 e 8 são relevantes por mostrar a superioridade do Algoritmo Genético e também destacar a pertinência do reuso de soluções passadas. A geração de dados utilizando o Algoritmo Genético e reuso de soluções requereu em média

Tabela 5.5: Programa2: Valores para NE utilizando Algoritmo Genético

Caminho / Tentativa	1	2	3	4	5	6	7	8	9	10
1	71	276	395	5	331	1	2	430	89	40
2	17	379	328	8	34	1	6	363	141	98
3	9	656	266	10	19	1	2	443	204	52
4	18	735	174	1	124	1	2	1216	77	20
5	54	474	251	14	172	1	7	1235	626	36
6	71	509	69	2	305	1	1	828	220	113
7	8	732	231	6	152	1	19	946	192	56
8	15	519	79	17	315	1	22	432	55	9
9	2	608	137	1	165	1	16	555	215	34
10	57	524	283	22	134	1	12	592	257	93
Totais:	322	5412	2213	86	1751	10	89	7040	2076	551
MEC:	32,2	541,2	221,3	8,6	175,1	1,0	8,9	704,0	207,6	55,1
MEP:	195,5									

Tabela 5.6: Programa2: Valores para NE utilizando Busca Aleatória

Caminho / Tentativa	1	2	3	4	5	6	7	8	9	10
1	11	14286	2386	2	330	1	4	45000^{Fl}	277	7
2	13	22526	1457	7	1249	1	15	45000^{Fl}	885	2
3	49	45000^{Fl}	270	6	460	1	6	25110	774	54
4	23	16699	1191	14	219	1	2	45000^{Fl}	682	38
5	35	29940	497	13	217	1	5	45000^{Fl}	612	73
Totais:	137	128451	5801	42	2475	5	32	205110	3230	174
MEC:	27,4	25690,2	1160,2	8,4	495,0	1,0	6,4	41022,0	646,0	34,8
MEP:	6909,14									

500,3 e 253,3 execuções, respectivamente, para os caminhos 2 e 8. Utilizando somente o Algoritmo Genético estes valores foram, respectivamente, 541,2 e 704,0. Isto revela uma inversão em relação à complexidade aparente dos dois problemas. Isto ocorre pois o reuso de soluções afeta de forma significativa a geração de dados para o caminho 8 pois, nesta ocasião, o esforço computacional dispendido na geração de dados para os sete caminhos anteriores gerou informações efetivamente úteis na base de soluções e permitiu a recuperação de dados de entrada consideravelmente próximos da solução do problema. Neste sentido cabe ainda destacar que os valores para estes caminhos na geração aleatória são coerentes com esta conclusão; isto é, o caminho 8 realmente requer mais esforço para a geração de dados em relação ao caminho 2.

Outra observação importante é que nos caminhos 3, 4, 6, 7, 9 e 10 a solução desejada foi descoberta diretamente na base de soluções (em todas as tentativas), sem que fosse necessária a realização da busca com o Algoritmo Genético. Isto representou um economia de 496,5 execuções no programa (na média das tentativas) em relação à busca sem o reuso de soluções.

A busca aleatória falhou em 5 das 50 tentativas. Nestas situações foi alcançado o valor limite para o número de execuções (45.000) sem que a solução fosse descoberta.

As falhas da busca aleatória ocorreram na geração de dados para os caminhos 2 e 8, já referidos. Nestes caminhos a razão entre o número de execuções na geração de dados utilizando a ferramenta e a geração aleatória foi de 88,53. Esta razão só não foi maior devido à limitação do número máximo de execuções em 45.000 (nestes casos a geração aleatória sem a limitação certamente superaria este valor — em média — para que a solução fosse encontrada).

É necessário destacar que o efeito da limitação do número máximo de execuções é o de reduzir o valor dos indicadores de eficiência 1 e 2; isto é, esta limitação tende a reduzir o desempenho aparente da ferramenta, tornando-o mais próximo da busca aleatória. Isto ocorre porque, para todos os casos (todos os programas e caminhos), a busca aleatória tende a atingir o limite máximo de número de execuções com uma freqüência muito maior do que nos modos de execução **i** e **ii** (utilizando Algoritmo Genético e Reuso de Soluções ou Algoritmo Genético apenas, respectivamente). Assim, na computação dos valores MEC e MEP com a busca aleatória, tende-se a obter valores menores do que os que seriam necessários se essa limitação não existisse.

Com base na análise anterior, pode-se afirmar que esta forma de cálculo para os indicadores de eficiência pode previlegiar a busca aleatória em detrimento da realizada pela ferramenta, mas o contrário não ocorrerá. Pode-se dizer, então, que estes indicadores fornecem uma visão conservadora e segura do desempenho da ferramenta. Isto confirma a idéia intuitiva de que, para certos caminhos que exigem valores de entrada muito particulares para serem executados, a geração aleatória (baseada na tentativa e erro e sem direcionamento da busca) tende a exigir um número de execuções e um tempo exorbitantes

Tabela 5.7: Programa3: Valores para NE utilizando Algoritmo Genético e Reuso de Soluções

Caminho / Tentativa	1	2	3	4
1	7203	1426	1252	1^{EB}
2	10166	709	2616	435
3	4191	1509	997	1^{EB}
4	3984	547	1006	1^{EB}
5	6580	1	1845	292
6	11095	9903	19460^{Pne}	99
7	1976	1839	976	15
8	5294	824	3525	1^{EB}
9	3617	18483^{Pne}	12710	1^{EB}
10	10499	7329	18242^{Pne}	840
Totais:	64605	42570	62629	1394
MEC:	6405,50	4270,00	6262,90	139,40
MEP:	4269,45			

para que dados de entrada adequados sejam encontrados.

5.2.3 Programa 3

Este programa recebe como entrada três variáveis globais: um vetor de quatro posições do tipo caracter ("string") e três variáveis do tipo caráter. Os predicados envolvem comparações de variáveis incluindo igualdades e desigualdades ($ch1 == 'a'$ e $ch3 < 'c'$). É utilizado um predicado realizando a comparação do "string" com o nome constante "test" ($!strcmp(nom, "test")$).

Interface informada à ferramenta:

$c1$: tipo "char";

$c2$: tipo "char";

$c3$: tipo "char";

nome: tipo "char[5]" ("string" formado de quatro símbolos) [1].

Indicadores de Eficiência:

[1]Na linguagem C vetores de n posições devem ser declarados com dimensão $(n+1)$.

Tabela 5.8: Programa3: Valores para NE utilizando Algoritmo Genético

Caminho / Tentativa	1	2	3	4
1	4515	8375	45000^{PneFl}	45000^{PneFl}
2	14988	5131	33346^{Pne}	32108^{Pne}
3	6685	9862	6911	41000^{Pne}
4	10101	26110^{Pne}	12300	15026
5	5323	3403	12109	18340
6	13211	8125	19241	5998
7	9203	5240	45000^{PneFl}	33101^{Pne}
8	2830	29030^{Pne}	13310	45000^{PneFl}
9	4115	9501	45000^{PneFl}	19015
10	3130	7712	9315	17376
Totais:	74101	112489	241532	271964
MEC:	7410,10	11248,90	24153,20	27196,40
MEP:	17502,15			

Tabela 5.9: Programa3: Valores para NE utilizando Busca Aleatória

Caminho / Tentativa	1	2	3	4
1	45000^{Fl}	45000^{Fl}	45000^{Fl}	45000^{Fl}
2	45000^{Fl}	45000^{Fl}	45000^{Fl}	45000^{Fl}
3	45000^{Fl}	45000^{Fl}	45000^{Fl}	45000^{Fl}
4	45000^{Fl}	45000^{Fl}	45000^{Fl}	45000^{Fl}
5	45000^{Fl}	45000^{Fl}	45000^{Fl}	45000^{Fl}
Totais:	225000	225000	225000	225000
MEC:	45000,00	45000,00	45000,00	45000,00
MEP:	45000,00			

1) $\frac{MEPiii}{MEPi} = 10,54$;

2) $\frac{MEPiii}{MEPii} = 4,10$;

3) $\frac{MEPii}{MEPi} = 2,57$;

Este caso permite a avaliação da geração de dados para programas que manipulem caracteres e "strings". A inserção do predicado que requer que o "string" *nom* seja igual a "test" pode ser visto como o pior caso no tratamento deste tipo de variável. Isto representa descobrir um nome específico em 128^4 possibilidades - considerando que cada posição do "string" pode assumir um dos 128 valores da tabela ASCII.

Obteve-se neste caso um desempenho pior em relação à obtida para outros programas. Apesar do valor *1)* ser significativo obteve-se na utilização do Algoritmo Genético sem o reuso de soluções (modo de execução ii) uma taxa de falhas da busca relativamente alta (cinco falhas em quarenta tentativas). As falhas ocorreram sobretudo na geração de dados de entrada para o caminho 4. Para executá-lo é necessário que as variáveis tenham os seguintes valores: $nom =$ *"test"*, $ch1 = 'a'$, $ch2 = 'b'$ e $ch3 < 'c'$. Verificou-se que, em geral, seria requerido um número maior de tentativas para que estes valores fossem encontrados. Entretanto, em todos os casos a busca foi encerrada tendo na população um grande número de dados de entrada próximos à solução (exemplos: *test#e7, testa_a e testNa/*). Por outro lado, na geração aleatória, houve falha em todos os casos. Provavelmente seria necessário um número expressivamente maior de tentativas para que soluções fossem descobertas. Em relação ao caminho 1 a geração aleatória foi reaplicada com limite de execuções de 100.000, em cinco tentativas obtiveram-se cinco falhas da busca.

Observou-se em diversas situações ausência de progresso persistente quando o "string" *"test_"* era encontrado. 0 símbolo *"_"* que substitui o *"a"* desejado (para execução dos caminhos 2, 3 e 4) tem código *ASCII* 95 enquanto o símbolo *"a"* tem código *ASCII* 97. Isto faz com que este "string" esteja relacionado com um alto nível de ajuste por estar próximo da solução. No entanto, na codificação binária os símbolos *"a"* e *"_"* são representados respectivamente por 1100001 e 1011111. A grande diferença entre estas representações explica a dificuldade do Algoritmo Genético em encontrar a solução desejada. Este problema, associado à codificação binária dos valores (chamado de "Hamming Cliffs"), tende a ser minimizado com a utilização de *código gray*. Esta possibilidade deve ser considerada para aperfeiçoamentos futuros visto que, nesta codificação, valores inteiros próximos possuem representações similares, aspecto que constatou-se ser importante no desempenho da busca.

5.2.4 Programa 4

Trata-se do programa *find* que recebe como entrada: o número de elementos de um vetor de inteiros; a posição do elemento a ser utilizado como "pivô" e um vetor de inteiros com

o número de elementos especificado. A partir desses valores, é gerado um vetor de saída com o mesmo número de elementos do vetor de entrada no qual todos elementos menores ou iguais ao valor do elemento pivô são posicionados à esquerda deste. Os elementos cujos valores forem maiores que o elemento pivô são posicionados à direita do mesmo.

Este programa permite a validação de aspectos importantes da ferramenta não considerados nos programas anteriores. Pode-se destacar o tratamento de variáveis do tipo vetor de números inteiros, tanto como valores de entrada para o módulo em teste, quanto nos predicados do programa (exemplo, predicado do ramo (14,15): $while(a[i] < a[f]))$. Tem-se também predicados do tipo lógico como: $if(!b)$. Outra particularidade refere-se à questão dos laços infinitos. Caso o valor do pivô seja superior ao número de elementos do vetor o programa "entra em loop infinito". O tratamento definido para este caso mostrou-se eficaz.

Note-se que tem-se neste programa a situação de "interface não definida estaticamente" descrita na Seção 4.4.1. Isto ocorre porque o número de elementos do vetor pode variar em cada execução, sendo este próprio número um valor de entrada. O tratamento feito consiste em definir o número de variáveis de entrada considerando um limite superior. Exemplificando: é informado à ferramenta que o vetor será de 15 elementos, caso a variável de entrada gerada para este número numa execução seja 10, as 5 restantes geradas pelo Algoritmo Genético serão desprezadas e não influirão na execução.

Interface informada à ferramenta:

n: tipo "int"; ordem de grandeza = 15;

p: tipo "int"; ordem de grandeza = 15;

a: tipo "int[16]" (vetor de inteiros com 15 posições); ordem de grandeza de cada elemento = 15.

Indicadores de Eficiência:

1) $\frac{MEPiii}{MEPi} = 6,06$;

2) $\frac{MEPiii}{MEPii} = 5,10$;

3) $\frac{MEPii}{MEPi} = 1,19$;

Este programa permitiu avaliar situações próximas do pior caso na utilização da ferramenta.

Predicados do tipo $(while(a[i] < a[f]))$ são de difícil tratamento. Nestes casos variações das variáveis de entrada afetam não só os valores associados às variáveis do predicado, mas também quais são as variáveis consideradas, visto que, se $i \neq j$, então $a[i]$ e $a[j]$ representam variáveis distintas.

Tabela 5.10: Programa4: Valores para NE utilizando Algoritmo Genético e Reuso de Soluções

Caminho / Tentativa	1	2	3	4	5
1	14080	664	226	2210	903
2	9653	230	511	53253	1^{EB}
3	13205	1^{EB}	352	12157	633
4	34808	353	2997	7127	874
5	11608	1^{EB}	1^{EB}	7307	166
6	4181	335	235	26500	237
7	23068	1^{EB}	1^{EB}	24370	1^{EB}
8	16215	123	1286	18320	463
9	8706	296	774	9381	212
10	21401	1^{EB}	1082	12130	285
Totais:	156925	2005	7465	172755	3775
MEC:	15692,50	200,50	746,50	17275,50	377,50
MEP:	6858,50				

Tabela 5.11: Programa4: Valores para NE utilizando Algoritmo Genético

Caminho / Tentativa	1	2	3	4	5
1	8582	1585	1506	44011	434
2	21775	1097	470	15628	2379
3	10480	880	1108	17631	1137
4	7341	1569	1823	12784	2828
5	3562	559	232	4005	1021
6	58778	322	927	2323	920
7	14816	1108	1248	36005	273
8	9210	730	822	38586	560
9	4022	1210	531	8095	920
10	5103	381	735	50069	1487
Totais:	143669	13518	9402	229137	11959
MEC:	14366,90	1351,80	940,20	22913,70	1195,90
MEP:	8153,70				

Tabela 5.12: Programa4: Valores para NE utilizando Busca Aleatória

Caminho / Tentativa	1	2	3	4	5
1	100000^{Fl}	1841	3666	100000^{Fl}	2211
2	100000^{Fl}	586	2702	100000^{Fl}	3608
3	100000^{Fl}	2578	2007	100000^{Fl}	817
4	100000^{Fl}	1738	729	100000^{Fl}	8971
5	100000^{Fl}	922	1936	100000^{Fl}	6434
Totais:	500000	7665	11040	500000^{Fl}	22041
MEC:	100000,00	1533,00	2208,00	100000,00	4408,20
MEP:	41589,84				

Como a primeira variável de entrada do programa determina o número de variáveis necessárias em cada execução, tem-se situações em que nem todas as variáveis codificadas na solução candidata são aplicadas no programa. Isto tende a gerar problemas pois não é informado ao Algoritmo Genético quais variáveis de entrada realmente influíram na execução e, portanto, na qualidade da solução candidata avaliada. Isto representa uma potencial fonte de discrepâncias por dificultar a identificação dos "blocos de construção" e tende a decrementar o desempenho do Algoritmo Genético.

Além destes aspectos, a existência de diversos laços permitiu a avaliação da ferramenta na geração de dados para caminhos longos; como exemplo, o caminho 1 possui 119 nós. A necessidade da satisfação de um grande número de predicados para a execução do caminho é um fator que comumente torna este problema bastante complexo — inclusive para geração manual de dados de teste.

A análise da evolução da busca permitiu identificar uma situação indesejável. Com certa frequência algumas soluções de ajuste muito superior à média dominavam prematuramente a população de soluções. Este fato levou o Algoritmo Genético a explorar durante várias gerações valores de entrada incompatíveis com os necessários para a execução dos caminhos. Esta situação ocorre quando um certo dado de entrada possui valores corretos para um conjunto de predicados do caminho, mas que impossibilitam a satisfação de algum predicado existente ao longo do mesmo. Caso este dado de entrada seja descoberto em estágios inicias da busca, existe a tendência de que as soluções caminhem em direção a valores sub-ótimos, até que soluções melhores que estas sub-ótimas sejam descobertas. A adoção da idéia de "fitness scaling" tende a minimizar este problema e merece ser considerada para aprimoramentos futuros.

5.2.5 Heurística de Identificação Dinâmica de Não Executabilidade: Análise de Resultados

No experimento conduzido foi registrado sempre que identificada a situação de potencial não executabilidade do caminho pretendido. Tendo em mente que todos os caminhos são executáveis tais situações podem ser vistas como erros de avaliação da heurística.

Considerando a geração de dados baseada no Algoritmo Genético e no reuso de soluções verificou-se três situações de erro de avaliação. O número total de tentativas foi de 240 (incluindo todos os programas e caminhos). Portanto, no experimento realizado a taxa de erros foi de $\frac{3}{240}$, ou 1,25%.

É importante observar que os erros de avaliação ocorreram em caminhos do programa 3. Isto se deve à existência do predicado $(!strcmp(nom, "test"))$ que é de difícil satisfação utilizando a otimização de soluções. Particularmente neste caso o erro de avaliação foi substancialmente maior (7,50%). Podem contribuir para a minimização deste problema a adoção de codificação gray, assim como a introdução na equação de cálculo do valor NL de um termo que aumente este valor na presença de predicados envolvendo estas variáveis.

Este fato observado sugere que o modelamento da complexidade do problema da geração de dados de teste para executar caminhos apenas em função do número de predicados e de variáveis de entrada (como feito até então na literatura) não é suficientemente preciso. Foi observado que predicados envolvendo o operador relacional de igualdade ($==$) demandaram um esforço computacional significativamente maior do que no caso de predicados utilizando outros operadores ($<, \leq, >, \geq$, e $\neq$). Além disso, predicados que requerem igualdade entre dois "strings" tornaram difícil a geração de dados e influenciaram negativamente a precisão da heurística proposta.

Pode-se ponderar que um melhor ajuste dos parâmetros utilizados tende a propiciar um melhor desempenho. Durante o experimento tais parâmetros foram mantidos constantes nos seguintes valores: $\delta = 0,50$; $\Delta = 0,30$; $K1 = 30$ e $K2 = 3$. Uma pequena redução de δ e Δ, assim como o aumento de $K1$ e $K2$, tende a reduzir a taxa de erros de avaliação, permitindo uma maior confiança na corretude da mesma. A consideração dos tipos de predicados do caminho pretendido e das variáveis neles envolvidas pode ser útil no aprimoramento deste modelo, merecendo uma análise futura pormenorizada.

Adicionalmente, foram submetidos à geração de dados alguns caminhos não executáveis. Em todos os casos a identificação desta condição, realizada pela heurística, foi bem sucedida. Como exemplo, considere-se o caminho: 1 2 3 5 6 14 16 18 19 20 21 2 22 do Programa 4. Este caminho é não executável pois contém o *padrão de não executabilidade* sequência de nós: 1 2 3 5 (a variável b recebe o valor 0 no nó 1, impedindo que a condição $if(b)$ associada ao arco $(3, 5)$ seja satisfeita). Neste caso, nas 10 tentativas feitas, foram requeridas em média 19.325 execuções do programa para a identificação da potencial não executabilidade. A mensagem emitida pela ferramenta foi do seguinte tipo:

```
DataGen - Identificada situacao de possivel nao executabilidade
do caminho pretendido.

Numero da Geracao: 146
Numero da Tentativa: 14600

O ajuste medio da populacao teve progresso acumulado inferior a 0.30
e progresso sucessivo inferior a 0.50 nas ultimas 114.000000 geracoes.

Progresso acumulado em 114.000000 geracoes: 0.025000

Caminho Pretendido:  1  2  3  5  6  14  16  18  19  20  21  2  22

Numero de nos corretamente executados da melhor solucao: 3
Ultimo no corretamente executado: 3
Verifique executabilidade do arco ( 3 , 5 ) em relacao aos predicados
anteriores do caminho.

Interrompe a busca ? (s|n)
```

Verificou-se que o reuso de soluções passadas tende a gerar uma população inicial com soluções que executam o caminho pretendido até nós muito próximos do ramo não factível. Este fato comumente possibilita que a heurística identifique a potencial não executabilidade em um número menor de tentativas. Isto ocorre porque a ausência persistente de progresso da busca acontece antecipadamente, pois o custo da descoberta de dados de entrada que atingem o predicado em questão é minimizado com o uso do conhecimento presente na base. Não foi conduzido, entretanto, um procedimento de avaliação no nível de redução do número de tentativas requeridas para a identificação de potencial não executabilidade neste caso.

5.2.6 Sumário dos Resultados

O experimento realizado permitiu uma avaliação inicial da ferramenta proposta. Foram utilizados quatro programas com características diferentes de tipos de variáveis tratadas, predicados e estruturas de controle. Foram selecionados ao todo vinte e quatro caminhos executáveis nesses programas. Foram definidos três modos de execução: utilizando o Algoritmo Genético com o reuso de soluções passadas; o Algoritmo Genético sem este reuso e uma busca aleatória. Para cada caminho e modo de execução foram feitas diversas tentativas de geração de dados de teste (um total de 600 tentativas). Em cada tentativa foi registrado o número de execuções do programa necessárias para que algum dado de entrada que executa o caminho fosse obtido (um total de 4.274.917 execuções).

Situações particulares foram discutidas na seção anterior, onde destacaram-se os pontos posistivos e negativos observados. A seguir são sumarizados e analisados os resultados

alganҫados no geral.

Custo e Eficácia na Geração Automática de Dados de Teste

A idéia de *eficácia* na automação da geração de dados de teste pode ser relacionada ao nível de sucesso obtido na geração de dados sem a intervenção do testador. Isto é, uma ferramenta para geração automática de dados de teste é eficaz na medida em que é capaz de gerar os dados de teste necessários para que os requisitos de teste sejam exercitados sem auxílio externo.

A *eficiência* na geração automática de dados de teste relaciona-se ao custo computacional necessário para que estes dados de teste sejam gerados.

Naturalmente deseja-se maximizar a eficácia e a eficiência neste processo. Isto implica, neste contexto, em gerar automaticamente dados de entrada que executem os caminhos pretendidos, com o máximo nível de sucesso. Ao mesmo tempo almeja-se que estes dados de entrada sejam gerados com o mínimo custo computacional.

Sob esta ótica são analisados a seguir os resultados apurados.

Resultados Obtidos Relacionados ao Custo e à Eficácia na Utilização da Ferramenta

Considerando-se todos os programas, caminhos e tentativas foram computados os valores médios dos indicadores de eficiência:

1) $\frac{MEPiii}{MEPi} = 8,34$;

2) $\frac{MEPiii}{MEPii} = 7,42$;

3) $\frac{MEPii}{MEPi} = 1,12$;

O uso da ferramenta permitiu que os dados de entrada desejados fossem obtidos com um número de execuções do programa $8,34$ vezes menor do que na geração aleatória. Utilizando apenas o Algoritmo Genético este valor foi de $7,42$. O reuso de soluções passadas permitiu um número de execuções 12% menor.

Utilizando o *Algoritmo Genético e o Reuso de Soluções Passadas* foram obtidos os dados de entradas que executam os caminhos pretendidos em todas as 240 tentativas (100% de acerto). Neste caso foram requeridas em média $2.831,49$ execuções dos programas, equivalentes a aproximadamente 8 minutos de busca.

Utilizando o *Algoritmo Genético apenas* foram obtidos os dados de entradas que executam os caminhos pretendidos em 235 das 240 tentativas ($97,92\%$ de acerto). Neste caso

foram requeridas em média 3.180, 11 execuções dos programas, equivalentes a aproximadamente 9 minutos de busca.

Utilizando a *Busca Aleatória* foram obtidos os dados de entradas que executam os caminhos pretendidos em 85 das 120 tentativas (70, 83% de acerto). Neste caso foram requeridas em média 23.601, 11 execuções dos programas, equivalentes a aproximadamente 67 minutos de busca.

Foi possível observar que a geração aleatória de dados é eficaz para alguns caminhos selecionados. Tais caminhos caracterizam-se por estarem associados a grandes sub-regiões do domínio de entrada do programa. Nestas situações é possível encontrar, após poucas tentativas, os valores de entrada pertencentes a estas sub-regiões, fazendo com que a geração aleatória seja aplicável. O caminho 1 do *programa 1* é um exemplo: para executá-lo é suficiente descobrir valores para as variáveis tipo "float" z e y tais que $z > y$. A sub-região definida por este predicado é representativa em relação ao domínio de entrada do programa.

Em outros casos os predicados existentes ao longo do caminho estabelecem restrições severas no domínio de entrada, fazendo com que apenas dados com características extremamente específicas executem-nos. Nestes casos a chance de que sejam gerados aleatoriamente valores para execução dos caminhos são pequenas, mesmo com um grande número de tentativas. Isto inviabiliza a utilização da geração aleatória. São exemplos desta situação todos os caminhos do *programa 3* e os caminhos 1 e 4 do *programa 4*. Exemplificando: para que o caminho 1 do programa 4 seja executado os predicados $(a[f] < a[j])$ e $(a[i] < a[f])$ (arcos $(16, 17)$ e $(14, 15)$ respectivamente) nunca devem ser satisfeitos, além do requisito de que todos os demais predicados existentes no caminho (são 119 nós e aproximadamente 100 predicados ao todo) sejam avaliados segundo o desejado. Valores de entrada que executam tal caminho possuem as seguintes características: $n = 10$, $p = 6$, $a[i] = K \; \forall \; i = 1 \; ... \; 10$, sendo K algum valor inteiro constante.

Considerando esta classe de caminhos, para a qual a geração de dados é não trivial, a razão entre o custo utizando a ferramenta e a geração aleatória tende a ser muito expressiva. A geração aleatória é proibitiva nesses casos. Este fato não se refletiu mais intensamente nos indicadores de eficiência apurados devido à limitação do número máximo de execuções do programa considerado. O efeito desta limitação foi de aproximar os resultados da geração ulilizando a ferramenta aos da busca aleatória.

Impacto do Reuso de Soluções Passadas no Custo e na Eficácia da Geração de Dados de Teste

O *Reuso de Soluções Passadas* permitiu uma redução de 12% em média do número de execuções do programa. Um aspecto significativo foi que conseguiu-se 100% de acerto com o reuso de soluções e o Algoritmo Genético, contra 97, 92% de acerto utilizando apenas

o Algoritmo Genético. Isto ocorreu porque o fato de a população inicial do Algoritmo Genético conter soluções "próximas" das buscadas tende a reduzir o número de execuções necessárias para que o dado de entrada que executa o caminho pretendido seja descoberto, aumentado assim a chance de sucesso antes do limite superior estabelecido para este número.

Comumente, à medida que os caminhos pretendidos são considerados pela ferramenta, tem-se um aumento do número de casos presentes na base de soluções. Deste modo, o impacto do reuso de soluções passadas na redução do custo do teste tende a ser maior na medida em que for maior o número de caminhos pretendidos para o teste.

Um aspecto negativo do reuso de soluções no modelo utilizado é que o conhecimento adquirido no teste de um determinado programa é útil apenas na geração de dados para elementos requeridos deste programa. Idealmente este conhecimento deveria ser aplicável para outros programas e critérios de teste. Desenvolvimentos futuros do reuso de soluções passadas devem considerar esta possibilidade.

Em geral o tempo necessário para a recuperação de soluções da base foi pouco significativo. Na maioria dos casos este tempo foi de alguns segundos. Entretanto, em situações extremas (caminhos longos, muitas variáveis de entrada e bases muito grandes) este tempo tende a aumentar. Aprimoramentos futuros podem incorporar estatégias mais eficientes para a busca dos casos da base de soluções, assim como realizar a inserção seletiva de casos, reduzindo assim a redundância dos casos apreendidos e o tamanho da base.

Eficácia da Heurística de Identificação Dinâmica de Potencial Não Executabilidade

A *heurística de identificação dinâmica de potencial não executabilidade* quando na utilização do Algoritmo Genético com o reuso de soluções passadas gerou erros de avaliação em 1,25% dos casos. Nestas situações caminhos executáveis foram identificados erroneamente como não executáveis. As tentativas de geração de dados para caminhos não executáveis resultaram, em todos os casos, na correta identificação desta condição pela heurística.

Foi utilizado um *modelo de complexidade para geração de dados de teste* considerando o número de nós do caminho pretendido e o número de variáveis de entrada do programa. A idéia é que este modelo permita a sintonia da heurística a partir da complexidade estimada do problema. Esta sintonia é essencial para que seja feita corretamente a distinção entre "predicados difíceis de serem satisfeitos" e "predicados não factíveis" nos caminhos. Os resultados obtidos sugerem qua a incorporação do tipo de variáveis tratadas e tipo de predicados existentes permitiriam o aprimoramento deste modelo.

A informação do último predicado corretamente tomado no caminho pretendido, dentre todas as soluções investigadas na busca, pode auxiliar a confirmação da não executabilidade. A análise das condições e das variáveis envolvidas neste predicado tende a reduzir

o escopo dos comandos do programa a serem investigados para a identificação da causa da não executabilidade. A computação do "slice" deste predicado permitiria esta redução de escopo. É possível também aplicar análise estática de fluxo de dados para selecionar automaticamente seqüências de nós anteriores ao predicado, que potencializem a chance de satisfação do mesmo.

Adicionalmente, os dados de entrada presentes nas últimas populações anteriores à identificação podem permitir a abstração de características genéricas desses dados, possivelmente associadas a situações não factíveis tendo em vista a especificação do programa.

É importante destacar que os resultados descritos nesta seção não são generalizáveis. Mesmo considerando o tratamento de diferentes tipos de variáveis, predicados e estruturas nos programas utilizados no experimento, não é possivel garantir os mesmos resultados para outros programas. A condução de mais estudos pode permitir a obtenção de resultados mais conclusivos e a identificação de possíveis melhorias na ferramenta.

Capítulo 6

Conclusão e Trabalhos Futuros

Todo inventor sempre é conseqüência de seu tempo e ambiente. Sua criatividade deriva das necessidades que foram criadas antes dele e baseia-se nas possibilidades que existem fora dele. É por isso que observamos uma continuidade rigorosa no desenvolvimento histórico da tecnologia e da ciência. Nenhuma invenção ou descoberta científica aparece antes de serem criadas as condições materiais e psicológicas necessárias para o seu surgimento. A criatividade é um processo historicamente contínuo em que cada forma seguinte é determinada pelas precedentes.

Lev Vygotsky

Neste capítulo retomam-se o tema e o objetivo deste trabalho. São sumarizadas as soluções propostas e os resultados obtidos. São destacados também aspectos que distingüem este trabalho dos existentes na literatura.

6.1 Conclusão

Esta seção resume o problema da geração de dados de teste e da detecção de não executabilidade de caminhos no teste estrutural de software. É retomado o objetivo inicial do trabalho e é descrita a abordagem utilizada assim como os conceitos principais a ela relacionados. São também sumarizados os resultados alcançados.

6.1.1 O Problema

Nesta dissertação foram abordados a automação da atividade de geração de dados de teste e da detecção de não executabilidade de caminhos no teste estrutural de software; tarefas comumente feitas de forma manual, que exigem do testador grande esforço mental e conhecimentos específicos dos critérios de teste utilizados.

Determinação de dados de teste e de não executabilidade são indecidíveis em situações genéricas, sendo tratadas invariavelmente de forma parcial; isto é, estabelecem-se soluções aplicáveis apenas às classes específicas de programas que respeitem pré-requisitos determinados [Cla76, FW88]. A não satisfação desses pré-requisitos implica a necessidade da intervenção humana.

Pesquisas visando à automação da geração de dados e da identificação de elementos requeridos não executáveis podem ser encontradas na literatura. Soluções, mesmo que parciais, são de grande valor, pois aumentam o nível de automação da atividade de teste e contribuem para o aumento de qualidade e a diminuição do custo desta atividade. Conforme destacado por Graham [Gra94], "na medida em que o teste se tornar mais eficiente e mais automatizado, ele se tornará ainda mais desafiador, criativo e divertido".

Neste cenário foi delineado o objetivo de desenvolver e implementar uma ferramenta que auxilie o testador na tarefa de gerar dados de teste e de identificar caminhos não executáveis no contexto da Ferramenta *POKE-TOOL* [Cha91] e dos critérios *Potenciais Usos* [Mal91]. Adicionalmente, tomou-se como diretriz elaborar soluções que reduzam a necessidade de interferência humana no processo de geração de dados de teste e que sejam aplicáveis para a maioria dos programas. Neste sentido, buscou-se elaborar soluções que fossem robustas e compatíveis com o tratamento de chamadas de funções; variáveis de diferentes tipos, incluindo "strings" e variáveis compostas; e com programas contendo comandos de seleção e de repetição.

6.1.2 A Abordagem

O problema da geração de dados de teste pode ser decomposto em dois outros: *i*) seleção de um caminho no programa que cubra cada elemento requerido pelo critério; e *ii*) geração de valores para as variáveis de entrada do programa tais que um caminho selecionado seja executado [Kor90]. Este trabalho abordou o item *ii*. Para diversos critérios relacionados à técnica estrutural de teste é possível selecionar caminhos que, quando executados, exercitam os elementos requeridos pelos critérios (excetuando naturalmente os elementos não executáveis). Portanto, a solução do problema *ii* é genérica para boa parte dos critérios estruturais, incluindo todos os baseados em análise de fluxo de dados.

A partir de modelos desenvolvidos foi feita a implementação de uma ferramenta de geração automática de dados de teste e identificação de potencial não executabilidade. A ferramenta integra conceitos consolidados na literatura apoiando-se, essencialmente, na *técnica dinâmica* [MS76, Kor90, GN97]; na otimização utilizando *Algoritmos Genéticos* [Gol89, SM94, Zbi96] e no reuso de soluções passadas relacionado ao paradigma do *Raciocínio Baseado em Casos* [Kol93, AE94, Lea96].

O papel do Algoritmo Genético é manipular os valores de entrada do programa a fim de fazer com que o caminho pretendido seja executado. Algoritmos Genéticos são

extremamente adequados ao problema da geração de dados de teste devido à sua elevada eficácia em descobrir soluções em espaços de busca complexos, incluindo problemas não lineares, multimodais e descontínuos; aspecto importante, dado que predicados de programas podem apresentar tais características.

O paradigma *Raciocínio Baseado em Casos* permite a recuperação na *Base de Soluções* de dados de entrada que executaram anteriormente caminhos similares ao pretendido. Isto torna possível que o Algoritmo Genético inicie a busca com uma população de soluções próximas da solução do problema ou, eventualmete, descobri-la diretamente na base. Esta abordagem tem o potencial de reduzir o custo e aumentar o nível de sucesso da geração automática de dados.

Devido à elevada eficácia do Algoritmo Genético na busca dos dados de entrada que executem os caminhos pretendidos, foi observada uma forte correlação entre a ausência persistente de progresso da busca com o fato do caminho pretendido ser não executável. Este fato permitiu um tratamento diferente dos exsitentes para a questão da *não executabilidade*. Este tratamento é feito pela *Heurística de Identificação Dinâmica de Potencial Não Executabilidade*, que consiste no monitoramento da dinâmica da busca feita pelo o Algoritmo Genético, permitindo a identificação de ausência persistente de progresso, provavelmente associada à não executabilidade do caminho pretendido. A heurística é aplicável indistintamente, independentemente do tamanho do programa, de tipos de variáveis envolvidas e de pré-requisitos sobre equações de caminhos tratáveis, limitações inerentes às abordagens existentes.

O *modelo de instrumentação* definido, assim como os aspectos considerados no cálculo da *função de predicado*, permitem a compatibilidade da ferramenta com diferentes tipos de variável e com as diversas estruturas presentes na linguagem C, tornando-a aplicável a problemas reais.

6.1.3 Os Resultados

Foi conduzido um experimento para avaliar o desempenho da ferramenta na geração de dados de teste para executar caminhos do programa. Avaliou-se também a influência do Reuso de Soluções Passadas na geração de dados e a eficácia da Heurística de Identificação Dinâmica de Potencial Não Executabilidade.

Em comparação com uma busca aleatória a ferramenta apresentou um maior nível de sucesso (100% contra 70,83% das tentativas de geração de dados bem sucedidas) e um custo sensivelmente menor (uma média de $2.831,49$ contra $23.601,11$ execuções dos programas necessárias para que os dados de entrada fossem encontrados). O Reuso de Soluções Passadas possibilitou uma redução média de $12,3\%$ no custo da geração de dados e um aumento de $2,08$ pontos percentuais no nível de sucesso. A heurística para tratamento da não executabilidade realizou a avaliação correta em $98,75\%$ das tentativas; erros de

avaliação ocorreram quando caminhos executáveis foram identificados erroneamente como não executáveis.

A análise dos resultados permitiu concluir que para caminhos mais complexos, que estabelecem restrições severas no domínio de entrada do programa, a geração aleatória é inviável. Nestes casos, a utilização da ferramenta pode possibilitar uma expressiva economia de tempo e de recursos computacionais. Além disso, analisando os programas e caminhos utilizados no experimento (sobretudo os que possuem um grande número de iterações em laços) é razoável supor que seriam requeridos tempos consideravelmente superiores para a geração manual desses dados, em boa parte dos casos.

6.2 Aspectos Importantes

Nesta seção são destacados aspectos que distingüem este trabalho dos existentes na literatura.

- O tratamento de variáveis dos tipos "string" e caráter e de predicados compostos é apresentado adicionalmente apenas por Gallagher [GN97]. Korel [Kor90] e Gupta [GMS98], assim como os demais autores, restringem o tratamento às variáveis numéricas e a predicados simples. Em [MS76] são consideradas apenas variáveis do tipo real. O tratamento de caracteres e "strings" requer a definição de como otimizar funções com este tipo de variável. A consideração de predicados compostos requisita um modelo de instrumentação compatível com a composição de valores para a função de predicado, questão não trivial. Deve-se destacar que em [GN97] não são relatados resultados de experimentos de validação utilizando programas reais.

- Não foram identificados trabalhos utilizando Algoritmos Genéticos aplicados com o objetivo de executar caminhos no programa. Em [JSE96] e [MMSC97, MMS98] utilizam-se Algoritmos Genéticos com o objetivo de minimizar funções associadas aos predicados, a fim de satisfazê-los.

  A utilização de Algoritmos Genéticos nesta dissertação visa a exercitar caminhos no programa. Isto torna a abordagem genérica e aplicável para a grande maioria de critérios associados à técnica estrutural. Adicionalmente, devido à *função de ajuste* definida, a busca tem um direcionamento preciso desde a primeira geração do Algoritmo Genético. Isto permite que a ferramenta seja utilizada também para complementar a cobertura obtida no teste. Isto no caso de ter sido feita a aplicação anterior de uma massa de dados pré-existente e a identificação de elementos requeridos ainda não exercitados.

- Nenhum dos trabalhos existentes utiliza informações anteriores do processo de geração de dados de teste para aprimorar a busca pelo dado de entrada que executa

o caminho pretendido. Também não existem trabalhos aplicando o paradigma do Raciocínio Baseado em Casos ao teste de software.

A idéia de utilizar Raciocínio Baseado em Casos no teste é diretamente aplicável para diversos trabalhos [MS76, Kor90, GN97, GMS98]. Isto é especialmente interessante visto que, dependendo da técnica de otimização utilizada, a escolha de pontos iniciais muito distantes da solução tende a causar falha na geração de dados [GN97].

- O tratamento da questão da não executabilidade, no contexto da geração automática de dados de teste, é essencialmente baseado em análise estática de fluxo de dados; processamento simbólico; provadores de teorema e análise numérica.

  Neste trabalho apresenta-se um tratamento alternativo a esta questão. É introduzida a abordagem baseada em informações dinâmicas, referentes à busca utilizando o Algoritmo Genético. Isto é possível pelo monitoramento do valor *média de ajuste* da população do Algoritmo Genético. que reflete fielmente a dinâmica da busca e permite a identificação segura de ausência persistente de progresso, provavelmente associada à não executabilidade do caminho pretendido.

- Neste trabalho foi utilizado o paradigma do Raciocínio Baseado em Casos visando recuperar soluções de problemas análogos, potencialmente próximos da solução do problema atual. O Algoritmo Genético utiliza estas soluções como uma região inicial para a busca da solução do problema atual. Nenhum dos trabalhos analisados referentes a esses dois conceitos (Algoritmos Genéticos e Raciocínio Baseado em Casos) faz menção à possibilidade de integrá-los para solução de problemas complexos. Observou-se neste trabalho o potencial do Raciocínio Baseado em Casos em permitir o aprimoramento da busca realizada pelo Algoritmo Genético e o potencial deste algoritmo como uma técnica para revisar e adaptar automaticamente soluções prévias inadequadas, recuperadas utilizando o Raciocínio Baseado em Casos.

6.3 Trabalhos Futuros

São destacadas a seguir possíveis linhas de pesquisa relacionadas ao tema desta dissertação, identificadas durante os estudos desenvolvidos. Procurou-se relatar tanto aprimoramentos possíveis nas soluções propostas quanto temas cujo estudo poderia gerar resultados úteis ao teste de software.

Aprimoramentos no Algoritmo Genético

Foi utilizado o Algoritmo Genético Simples definido por Holland [Gol89]. Em [Zbi96] são discutidos aspectos da utilização de Algoritmos Genéticos para satisfação de restrições e introduzidas idéias para o aprimoramento desses algoritmos para este problema. Um

estudo mais aprofundado desse tema pode permitir uma melhor adaptação do Algoritmo Genético à geração de dados de teste. Inicialmente devem ser analisadas a adoção de codificação "gray" e o "fitness scaling", cuja adequação foi identificada nos experimentos.

Em [GMS98] Gupta introduz uma abordagem dinâmica para a geração de dados de teste em que é derivada uma *representação linear aritmética* dos predicados. Caso o predicado seja linear é possível, partindo desta representação, determinar os dados de entrada que satisfazem-no. Esta idéia pode ser incorporada como um "operador genético especializado" que introduz na população do Algoritmo Genético uma solução supondo a linearidade de um dado predicado logo que este seja atingido. Se este fosse o caso, o predicado seria resolvido imediatamente; caso contrário, o Algoritmo Genético realizaria a otimização utilizando valores da *função de predicado* normalmente.

Uma outra linha consiste em aprimorar a implementação. A comunicação entre o Algoritmo Genético e o programa em execução é feita através de arquivos seqüenciais e a execução do programa em teste é feita pelo comando *system*, que interrompe o Algoritmo Genético e requer o gerenciamento de contextos. Uma possibilidade interessante consiste em utilizar comandos do UNIX para estabelecer uma comunicação mais eficiente. O comando *fork* poderia ser utilizado para criar um processo concorrente executando o programa em teste e um *pipe* poderia direcionar informações geradas pela instrumentação para o Algoritmo Genético. O sincronismo seria feito pelo sistema operacional de forma transparente. Isto certamente reduziria expressivamente o tempo necessário para a avaliação das soluções candidatas com a conseqüente diminuição do tempo médio de busca. Ainda nesta linha, deve ser considerada a possibilidade de incorporar no modelo a idéia de *processamento distribuído*. Isto poderia permitir a avaliação em paralelo de diversas soluções candidadas e uma diminuição ainda maior do tempo médio de busca.

É possível também rever o modelo de instrumentação de forma que as "pontas de prova de caminhos" sejam eliminadas, sendo a informação a elas associada, obtida pelas "pontas de prova de predicados". Isto também permitiria uma redução substancial do tempo de busca.

Aprimoramentos na Heurística de Identificação de Potencial Não Executabilidade

Esta heurística é adaptada automaticamente ao caminho particular em teste através de parâmetros variáveis. É utilizado o número de variáveis de entrada do programa e o número de nós do caminho para estimar a complexidade do problema de geração de dados. A consideração adicional dos tipos de operadores relacionais e tipos de variáveis envolvidas nos predicados poderia tornar esta estimativa mais precisa. Seria possível, por exemplo, permitir uma maior ausência de progresso na busca, sem a identificação da potencial não executabilidade, quando o predicado atingido pelas soluções do Algoritmo Genético: envolver o operador relacional "="; for composto com operadores lógicos AND;

ou ainda, fazer comparações de "strings". Verificou-se que nesses casos a heurística tende a "confundir" um predicado difícil de ser satisfeito com um que causa não executabilidade do caminho.

Atualmente quando a heurística identifica a possível não executabilidade do caminho pretendido o testador é informado do predicado "suspeito" e precisa intervir analisando o programa. Nesta situação poderia ser feita a seleção automática de um outro caminho para cobrir o elemento requerido e iniciada uma nova busca. Esta seleção poderia ser feita utilizando-se a "Chaining Approach" de Ferguson [FK96], que faz uma análise de dependência de dados para selecionar diferentes seqüências de nós anteriores ao predicado visando satisfazê-lo. Seria possível também computar o "slice" deste predicado a fim de restringir o escopo de análise para a confirmação da não executabilidade.

Seria interessante também a definição de um modelo *M(Custo,Complexidade)* $\Rightarrow$ *Confiabilidade* elaborado empiricamente relacionando: *Custo* computacional dispendido na busca e *Complexidade* estimada do problema de geração de dados; com a *Confiabilidade* associada à identificação da não executabilidade em um certo (caminho, programa), em uma determinada tentativa de geração de dados. Isto permitiria que, ao ser identificada a potencial não executabilidade, o testador fosse informado da confiabilidade desta identificação. Baseado nisso, poderia ser decidida a continuidade da busca até que esta identificação seja associada a um nível de confiabilidade superior.

Modelo Genérico Para a Geração Automática de Dados de Teste

A ferramenta foi desenvolvida para apoiar a aplicação de critérios de teste estrutural; entretanto, é possível adaptá-la para a geração de dados para diferentes critérios, associados a outras técnicas. Essencialmente, gerar dados para executar caminhos implica em resolver as restrições associadas aos predicados do caminho pretendido. Essas restrições representam funções das variáveis de entrada do programa. Independentemente da técnica de teste, gerar dados requer a satisfação de restrições associadas aos elementos requeridos pelos critérios. Visto deste modo, não existe distinção se estas restrições referem-se a um requisito funcional, estrutural, ou baseado em defeitos.

Exemplificando: para que um programa mutante seja "morto" no critério análise de mutantes é necessário satisfazer as restrições de alcançabilidade, necessidade e suficiência. A primeira restrição requer que sejam encontrados dados que satisfaçam predicados de forma a alcançar o comando do programa que sofreu a mutação. A segunda restrição pode ser vista como um "predicado adicional" a ser satisfeito, requerendo dados que provoquem um estado do programa mutante diferente do programa original. A terceira restrição requer que o mutante apresente uma saída diferente do programa original, sendo a solução desta restrição indecidível genericamente.

Considerando critérios funcionais, por exemplo, o "particionamento em classes de equivalência", é possível criar programas que sejam "representações executáveis" dos requisitos

de teste. Estes programas incorporariam nos seus predicados as restrições associadas às classes de equivalência; deste modo, gerar dados que executem caminhos nos mesmos, equivaleria a gerar dados que satisfazem classes funcionais requeridas por esse critério.

Neste contexto o tratamento de "mutantes equivalentes" ou de possíveis "classes funcionais não factíveis" poderia ser feito através da heurística dinâmica de identificação de potencial não executabilidade.

Este tratamento genérico requer a definição de modelos para tratar as restrições associadas aos requisitos de teste de diferentes técnicas uniformemente. Isto permitiria a geração automática de conjuntos de dados de teste com a propriedade de satisfazer diversos critérios, associados às diferentes técnicas. Este "macro conjunto de teste" poderia ser minimizado a fim de eliminar redundâncias, aspecto relacionado ao próximo tema de trabalho futuro.

Aplicação de Algoritmos Genéticos Para a Minimização do Conjunto de Dados de Teste

Ao se gerar dados de teste para satisfazer um determinado critério comumente o conjunto gerado não é o menor possível. O tamanho do conjunto de dados de teste tem uma relação direta com o custo do teste de regressão. Além disso, a minimização deste conjunto propicia a redução do custo de armazenamento de informações relativas ao teste, além de diminuir o impacto negativo da utilização de dados redundantes no custo do teste.

A minimização de conjuntos de dados de teste tem o objetivo de, a partir de um conjunto de casos de teste T, gerar um conjunto T', com o mesmo grau de adequação de T, e com um tamanho menor. A redução do tamanho do conjunto T é obtida pela eliminação de casos de teste redundantes e obsoletos [HSG93, OPV97]. Estudos indicam que esta redução não implica necessariamente em perda de eficácia no teste [WHLM98]. Este problema é np-completo; portanto, soluções existentes são heurísticas, visando a definição de algum sub-conjunto do conjunto de casos de teste original que mantenha o nível de cobertura desejado.

Uma linha de trabalho possível, consiste em modelar o problema da minimização do conjunto de dados de teste considerando o objetivo de maximizar o número de casos de teste redundantes ou obsoletos identificados no conjunto original e removê-los. Isto permitiria a aplicação do Algoritmo Genético. Nesta abordagem cada solução candidata da população do Algoritmo Genético representaria uma possível ordem para a aplicação dos dados de teste. O ajuste de cada solução candidata quantificaria a capacidade da solução em maximizar os dados de entrada redundantes eliminados do conjunto original, refletindo o número de dados redundantes descobertos ao aplicá-los na ordem codificada na solução. O objetivo seria o de descobrir uma ordem para aplicação desses dados tal que seja minimizado o número de dados necessários para atingir a cobertura de todos os elementos requeridos pelo critério de teste.

Esta abordagem traz semelhanças com a utilização de Algoritmos Genéticos para a solução do *Problema do Caixeiro Viajante (T.S.P.)*, já amplamente estudada [Zbi96]. A semelhança está na necessidade de representar, em ambos os casos, a solução candidata como um "caminho em grafo" ou uma "lista de adjacências" e aplicar operadores genéticos compatíveis, isto é, que não gerem soluções passando mais de uma vez na mesma cidade — ou analogamente executando mais de uma vez um caso de teste. Considerando a complexidade do problema e o fato de que as soluções atualmente disponíveis são heurísticas, com tempo de execução de ordem polinomial, identifica-se a utilização dos Algoritmos Genéticos como uma alternativa viável para obtenção de soluções melhores do que as obtidas com as heurísticas, mas com custo acessível.

Em consonância com o trabalho futuro imediatamente anterior, seria pertinente definir a "minimização do conjunto de dados de teste multi-critérios". Ao contrário das abordagens citadas, que identificam a redundância de dados com respeito a um único critério, seriam eliminados dados de teste redundantes com respeito a diversos critérios de teste, considerados simultaneamente.

Geração Automática de Dados Para o Teste de Integração

Ainda que os diversos módulos do software tenham sido adequadamente testados, não é possível afirmar que estes módulos funcionarão como o esperado quando integrados. Isto ocorre porque defeitos nas interfaces dos módulos podem estar presentes, fato que destaca a importância do teste de integração. Diversos trabalhos estenderam conceitos do teste estrutural de unidades para o teste de integração [HS80, LM90, Vil98]. Critérios de teste de integração baseados em análise de fluxo de dados enfocam as relações e interfaces entre os módulos e requerem, essencialmente, que sub-caminhos associados às dependências de dados inter-procedimentais sejam exercitados.

Uma possível abordagem para a geração automática de dados para o teste de integração é a de buscar dados de entrada que exercitem caminhos inter-procedimentais, selecionados pelos critérios de teste. O contexto seria o dos Critérios Potenciais Usos de Integração [Vil98] definidos por Vilela e o objetivo seria gerar automaticamente dados de teste para esses critérios.

Em relação ao estado atual da ferramenta seria necessária a adaptação do modelo de instrumentação definido. Os diversos programas a serem testados de forma integrada seriam instrumentados, visando permitir o monitoramento do fluxo de execução nesses programas e entre eles.

Geração Automática de Dados Para o Teste de Regressão

O teste de regressão é realizado em programas modificados para estabelecer confiança de que as mudanças realizadas foram corretas e que não afetaram as partes não alteradas do programa.

Este teste utiliza um conjunto de dados anterior, reaplicando-os totalmente ou de forma seletiva. Comumente esta seleção é baseada em informações sobre os códigos do programa original e da versão modificada [RH96]. O teste de regressão pode requerer também a geração da dados adicionais para exercitar elementos requeridos específicos, situação na qual é possível utilizar a ferramenta proposta nesta dissertação.

Este trabalho futuro deve considerar ferramenta de apoio ao teste de regressão definida por Granja [GJ97], que situa-se no contexto da POKE-TOOL e dos critérios Potenciais Usos.

Uma possibilidade interessante relaciona-se à proposta de Korel [KAY98]. Neste trabalho é proposta uma técnica para a geração automática de dados de teste visando revelar defeitos existentes em programas modificados. A idéia é gerar dados de teste para programas cuja funcionalidade não é alterada após a modificação. O objetivo é encontrar dados de teste que façam com que o programa original e o modificado apresentem (erroneamente) diferentes resultados; este problema é reduzido ao de gerar dados que executem um determinado comando do programa.

Raciocínio Baseado em Casos e Algoritmos Genéticos

Este trabalho consiste em aprofundar o estudo nas duas áreas procurando identificar aspectos conceituais comuns e estabelecer uma integração construtiva das mesmas, que propicie benefícios recíprocos. O objetivo é analisar a validade e a utilidade de se estender a integração de Algoritmos Genéticos com o paradigma do Raciocínio Baseado em Casos, realizada neste trabalho, para o tratamento de outros problemas.

Automação da Instrumentação do Programa em Teste

Nesta dissertação foi definido o modelo de instrumentação do programa que permite o monitoramento da execução. Este modelo visa a obtenção das informações necessárias ao direcionamento da busca pelos dados de entrada que executem os caminhos pretendidos.

Este trabalho consiste na implementação de uma ferramenta que, através de análise estática do código fonte, insira automaticamente as pontas de prova definidas nos locais determinados pelo modelo.

Realização de Experimentos Adicionais

Foi definido um procedimento para a validação das idéias propostas e conduzido um experimento de avaliação inicial da ferramenta. Este trabalho visa a condução de experimentos adicionais, utilizando uma maior variedade de programas, a fim de aprimorar a ferramenta e obter resultados mais conclusivos.

Bibliografia

[AE94] A. Aamodt and Plaza E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AICom — Artificial Intelligence Communications*, Vol. 7(1):39–59, 1994.

[BCM+95] P.M.S. Bueno, M.L. Chaim, J.C. Maldonado, M. Jino, and P.R.S. Vilela. *Manual do Usuário da POKE-TOOL.* Technical Report, DCA/FEEC/Unicamp, Campinas - SP, 1995.

[Bei95] B. Beizer. *Black-Box Testing Techiniques for Functional Testing of Software Systems.* John Wiley e Sons, Inc., New York, 1995.

[BEK75] R.S. Boyer, B. Elspas, and N.L. Karl. Select: A formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, Vol. 10(6):234–245, Junho 1975.

[BGS97] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *Software Engineering Notes*, Vol. No 22(6):361–377, Novembro 1997.

[BJ97] P.M. Bueno and M. Jino. *Geração Automática de Dados de Teste: Um Estudo Inicial.* Technical Report, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Dezembro 1997.

[BJVM97] P.M. Bueno, M. Jino, S.R Vergilio, and J.C. Maldonado. Uma proposta de extensão da ferramenta poke-tool para apoiar o tratamento de não executabilidade. In *Workshop do Projeto Validação e Teste de Sistemas de Operação*, pages 213–222. Águas de Lindóia-SP, Janeiro 1997.

[BM83] D.L. Bird and C.U. Munoz. Automatic generation of random self-checking test cases. *IBM System Journal*, Vol. 22(3):229–245, 1983.

[Cha91] M.L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados.* Master Thesis, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Abril 1991.

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

[CR83] L.A Clarke and D.J. Richardson. The application of error-sensitive testing strategies to debugging. In *Symposium of High-Level Debugging*, pages 45–52. ACM SIGSOFT/SIGPLAN, Maio 1983.

[Del93] M.E. Delamaro. *Proteum: Um ambiente de teste baseado na Análise de Mutantes.* Master Thesis, ICMSC-USP-São Carlos, São Carlos - SP, Brazil, Março 1993.

[DMLS78] R.A. De Millo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. C-11:34–41, Abril 1978.

[FB97] I. Forgács and A. Bertolino. Feasible teste path selection by principal slicing. *Software Engineering Notes*, Vol. No 22(6):378–394, Novembro 1997.

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

[Fra87] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., Outubro 1987.

[FW86] F.G. Frankl and E.J. Weyuker. Data flow testing in the presence of unexecutable paths. In *Proceedings of the Workshop on Software Testing*, pages 4–13. Computer Science Press, Banff - Canada, Julho 1986.

[FW88] F.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, Vol. SE-14(10):1483–1498, Outubro 1988.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *Software Engineering Notes*, Vol. No 23(2):53–62, Março 1998.

[GJ97] I. Granja and M. Jino. *Uma Ferramenta de Apoio ao Teste de Regressão.* Master Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Dezembro 1997.

[GMS98] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. *Foundations of Software Engineering*, Vol. No 11:231–244, Novembro 1998.

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

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc, University of Alabama, 1989.

[Gra94] D. R. Graham. Testing. In *Encyclopedia of Software Engineering*, pages 1330–1352. John Wiley e Sons, Inc. — John J. Marciniak Editor-in-Chief, N.Y., 1994.

[Ham98] D. Hamlet. What can we learn by testing a program. *Software Engineering Notes*, Vol. 23(2):50–53, Março 1998.

[Het73] B. Hetzel. *Program Test Methods.* Prentice-Hall, N. J., 1973.

[HH85] D. Hedley and M.A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of 8th. ICSE*, pages 259–266. UK, 1985.

[HL90] J.R. Horgan and S. London. *ATAC — Automatic Test Coverage Analysis for C Programs*, Junho 1990.

[How75] W.E. Howden. Methodology for the generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-24(5):554–559, Maio 1975.

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

[How78] W.E. Howden. Dissect — a symbolic evaluation and program testing system. *IEEE Transactions on Software Engineering*, Vol. SE-4(1):70–73, Janeiro 1978.

[HS80] M.J. Harrold and M.L. Soffa. Selecting and using data for integration testing. *IEEE Software*, Vol. 8(2):58–65, Março 1980.

[HSG93] M.J. Harrold, M.L. Soffa, and R. Gupta. A methodology for controlling the size of a test suite. *ACM Transactions on on Software Engineering and Methodology*, Vol. 2(3):270–285, Julho 1993.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE, New York, 1990.

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

[KAY98] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98). ACM Software Engineering Notes 23,2*, pages 143–152. ACM Press, Cleawater Beach, FL, USA, Março 1998.

[Kol93] J. L. Kolondner. *Case-Based Reasoning.* Morgan Kaufmann Publishers, 1993.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, Vol. SE-16(8):870–879, Agosto 1990.

[Kor96] B. Korel. Automated test data generation for programs with procedures. *ACM Transactions on Software Engineering and Methodology*, 5(1):209–215, Janeiro 1996.

[Lea96] D. B. Leake. *Case-Based Reasoning Experiences, Leassons and Future Directions.* AAAI Press — The MIT Press, Menlo Park, CA 94025, 1996. Collection: 17 articles.

[LK83] J.W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, Vol. SE-9(3):347–354, Maio 1983.

[LM90] U. Linnenkuger and M. Mullerburg. Test data selection criteria for (software) integration testing. In *Proceedings of the First International Conference on Systems Integration*, pages 709–717. IEEE Press, Morristown, New Jersey, 23-26, Abril 1990.

[Mal91] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Julho 1991.

[MCJ88] J.C. Maldonado, M.L. Chaim, and M. Jino. Seleção de casos de testes baseada nos critérios potenciais usos. In *II Simpósio Brasileiro de Engenharia de Software*, pages 24–35. Sociedade Brasileira de Computação - SBC, Canela - RS, Brazil, Outubro 1988.

[MCVJ91] J.C. Maldonado, M.L. Chaim, S.R. Vergilio, and M. Jino. Critérios potenciais usos: Uma contribuição para a atividade de garantia de qualidade de software. In *Workshop em Avaliação de Qualidade de Software.* COPPE/UFRJ, Rio de Janeiro - RJ, Maio 1991.

[MJM75] E.F. Miller Jr. and R.A. Melton. Automated generation of testcase datasets. *ACM SigPlan Notices*, Vol. 10(6):51–58, Junho 1975.

[MMS98] G. McGraw, C. C. Michael, and M. A. Schatz. *Generating Software Test Data by Evolution.* Technical Report RSTR-018-97-01, RST Corporation: Suite 250, 21515 Ridgetop Circle Sterling, CA 20166, Fevereiro 1998. (at http://www.rstcorp.com/paper-subject.html).

[MMSC97] C. C. Michael, G. E. McGraw, M. A. Schatz, and Walton C. C. *Genetic Algorithms for Dynamic Test-Data Generation.* Technical Report RSTR-003-97-11, RST Corporation: Suite 250, 21515 Ridgetop Circle Sterling, CA 20166, Maio 1997. (at http://www.rstcorp.com/paper-subject.html).

[MS76] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(3):223–226, Setembro 1976.

[MVCJ92] J.C. Maldonado, S.R. Vergilio, M.L. Chaim, and M. Jino. Critérios potenciais usos: Análise de aplicação de um benchmark. In *VI Simpósio Brasileiro de Engenharia de Software*, pages 357–371. Gramado-RS, Brazil, Novembro 1992.

[MYE79] G.J. MYERS. *The Art of Software Testing.* J. Wiley, 1979.

[MYV90] N. Malevris, D.F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, Vol. 32(2):115–118, Março 1990.

[Nta84] S.C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10(6):795–803, Novembro 1984.

[Off96] A.J. Offut. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, Janeiro 1996. (Clemsom University, SC 29634).

[OP95] A.J. Offut and J. Pan. The dynamic domain reduction procedure for test data generation. *Technical Report*, Junho 1995. (to appear).

[OPV97] A.J. Offut, J. Pan, and J.M. Voas. Procedures for reducing the size of coverage-based test sets. *Technical Report — Reliable Software Tecnologies*, 1997. (to appear).

[Per97] L.M. Peres. *Estudo de Estratégias de Seleção de Caminhos para Satisfação de Critérios de Teste Estrutural.* Master Thesis, DCA/FEEC/Unicamp, Campinas - SP, 1997. (em elaboração).

[PH97] A.M.A. Price and J.S. Herbert. Estratégia de geração de dados de teste baseada na análise simbólica e dinâmica do programa. In *XI Simpósio Brasileiro de Engenharia de Software*, pages 397–411. Fortaleza , Brazil, Novembro 1997.

[Pre97] R.B. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New-York, EUA, fourth edition, 1997.

[RH96] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Agosto 1996.

[RMB+95] M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood. *Genetic Algorithms and the Automatic Generation of Test Data.* Technical Report RR/95/195[EFoCS-19-95, University of Strathclyde. Glasgow G1 1XH, U.K., 1995.

[RSBC76] C.V. Ramamoorthy, F. H. Siu-Bun, and W.T. Chen. On automated generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):293–300, Dezembro 1976.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Abril 1985.

[SM94] M. Srinivas and Patnaik L. M. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, Junho 1994.

[Ver92] S.R. Vergilio. *Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas.* Master Thesis - DCA/FEEC/Unicamp, Campinas - SP, Brazil, Janeiro 1992.

[Ver97] S.R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Julho 1997.

[Vil98] P.R.S. Vilela. *Critérios Potenciais Usos de Integração: Definição e Análise.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, Abril 1998.

[VMJ92] S.R. Vergilio, J.C. Maldonado, and M. Jino. Caminhos não executáveis na automação das atividades de teste. In *VI Simpósio Brasileiro de Engenharia de Software*, pages 343–356. Gramado -RS, Brazil, Novembro 1992.

[Wey82] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, Vol. 25(4):465–470, Setembro 1982.

[WGS94] E.J. Weyuker, T. Goradea, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, Vol. SE-20(5):353–363, Maio 1994.

[WHLM98] W.E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience*, Vol. 28(4):347–369, Abril 1998.

[Zbi96] Michalewicz Zbigniew. *Genetic Algorithms + Data Strutures = Evolution Programs.* Springer Verlag, North Carolina - USA, 3rd edition, 1996.

Apêndice A

Listagem de Programas Instrumentados

Este apêndice mostra exemplos de programas instrumentados segundo o modelo definido no Capítulo 4 desta dissertação.

Programa 1:

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

FILE * error;
int nodecount;

void error_write(int no, float exp1, float exp2, char *oprel, char *oplog)
 {
   fprintf(error,"%d %f %f %s %s\n", no, exp1, exp2, oprel, oplog);
   nodecount++;
   if (nodecount > 10000) exit(1);
 }

main()
/*  1 */        {
                   FILE * path = fopen("path.tes","w");
                   static int printed_nodes = 0;
/*  1 */           float x, y, z, t, t2;
                   error = fopen("error.tes","w");
                   nodecount = 0;
                   ponta_de_prova(1);
/*  1 */           printf("%s\n\n","entre com x, y e z ");
/*  1 */           scanf("%f\n",&x);
/*  1 */           scanf("%f\n",&y);
/*  1 */           scanf("%f\n",&z);
/*  1 */           printf("%s%f\n","X: ",x);
```

```
/*  1 */            printf("%s%f\n","Y: ",y);
/*  1 */            printf("%s%f\n\n","Z: ",z);
                    error_write(1,0,0,"#","#");
/*  1 */            if(z > y)
/*  2 */            {
                       ponta_de_prova(2);
                       error_write(2,z,y,"<=","#");
/*  2 */               if(y > x)
/*  3 */               {
                          error_write(3,y,x,"<=","#");
                          ponta_de_prova(3);
/*  3 */                  printf("%s\n","z > y > x ");
/*  3 */                  t = x + y;
/*  3 */                  if(t < z)
/*  4 */                  {
                             error_write(4,t,z,">=","#");
                             ponta_de_prova(4);
/*  4 */                     printf("%s\n","z > x + y");
/*  4 */                     t2 = x * y;
/*  4 */                     if( ((t2 - z) <= 5) && ((t2 - z) >= 0))
/*  5 */                      {
/*  5 */                        error_write(5,t2 - z,5,">","||");
/*  5 */                        error_write(5,t2 - z,0,"<","#");
/*  5 */                        ponta_de_prova(5);
/*  5 */                        printf("%s\n","(x * y) - z <= 5 e (x * y) - z >= 0");
/*  5 */                      }
/*  6 */                     error_write(6,t2 - z,5,"<=","&&");
/*  6 */                     error_write(6,t2 - z,0,">=","#");
/*  6 */                     ponta_de_prova(6);
/*  6 */                  }
/*  7 */                 else
/*  7 */                  {
                             error_write(7,y,x,">","#");
                             ponta_de_prova(7);
/*  7 */                     printf("%s\n","z < = x + y");
/*  7 */                  }
                          ponta_de_prova(8);
                          error_write(8, 0, 0,"#","#");
/*  8 */               }
                       error_write(9,y,x,">","#");
                       ponta_de_prova(9);
/*  9 */            }
                    error_write(10,z,y,">","#");
                    ponta_de_prova(10);
                    fclose(error);
                    fclose(path);
/* 10 */        }
```

Programa 2:

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <math.h>

FILE * error;
int nodecount;

void error_write(int no, float exp1, float exp2, char *oprel, char *oplog)
 {
   fprintf(error,"%d %f %f %s %s\n", no, exp1, exp2, oprel, oplog);
   nodecount++;
   if (nodecount > 100000) exit(1);
 }


main()
/*  1 */        {
                   FILE * path = fopen("path.tes","w");
                   static int printed_nodes = 0;
/*  1 */           int n, d, q, r, t;
                   ponta_de_prova(1);
                   nodecount = 0;
                   error  = fopen("error.tes","w");
/*  1 */           printf("%s","Dividendo: ");
/*  1 */           scanf("%d\n",&n);
/*  1 */           printf("%s","Divisor: ");
/*  1 */           scanf("%d\n",&d);
/*  1 */           printf("\n");
/*  1 */           printf("%s%d\n","Dividendo: ",n);
/*  1 */           printf("%s%d\n","Divisor: ",d);
/*  1 */           printf("\n");
/*  1 */           error_write(1,0,0,"#","#");
/*  1 */           if(d != 0)
/*  2 */           {
                      ponta_de_prova(2);
/*  2 */              error_write(2,d,0,"==","#");
/*  2 */              if( (d > 0) && (n > 0) )
/*  3 */              {
                         ponta_de_prova(3);
/*  3 */                 error_write(3,d,0,"<=","||");
/*  3 */                 error_write(3,n,0,"<=","#");
/*  3 */                 q = 0;
/*  3 */                 r = n;
/*  3 */                 t = d;
/*  4 */                 while(r >= t)
/*  5 */                 {
/*  5 */                    error_write(4,0,0,"#","#");
/*  5 */                    error_write(5,r,t,"<","#");
                            ponta_de_prova(4);
                            ponta_de_prova(5);
/*  5 */                    t = t * 2;
/*  3 */                 }
/*  4 */                 error_write(4,0,0,"#","#");
/*  6 */                 error_write(6,r,t,">=","#");
                         ponta_de_prova(4);
```

```
/*  6 */                while(t != d)
/*  7 */                {
                           ponta_de_prova(6);
/*  6 */                   error_write(6,0,0,"#","#");
/*  7 */                   error_write(7,t,d,"==","#");
                           ponta_de_prova(7);
/*  7 */                   q = q * 2;
/*  7 */                   t = t / 2;
/*  7 */                   if(t <= r)
/*  8 */                   {
                              ponta_de_prova(8);
/*  8 */                      error_write(8,t,r,">","#");
/*  8 */                      r = r - t;
/*  8 */                      q = q + 1;
/*  8 */                   }
/*  9 */                   error_write(9,t,r,"<=","#");
                           ponta_de_prova(9);
/*  9 */                }
                           ponta_de_prova(6);
/*  6 */                   error_write(6,0,0,"#","#");
                           ponta_de_prova(10);
/* 10 */                   error_write(10,t,d,"!=","#");
/* 10 */                   printf("%s%d\n","Quociente: ",q);
/* 10 */                   printf("%s%d\n","Resto: ",r);
/* 10 */             }
/* 11 */             else
/* 11 */             {
                        ponta_de_prova(11);
/* 11 */                   error_write(11,d,0,">","&&");
/* 11 */                   error_write(11,n,0,">","#");
/* 11 */                   printf("%s\n","Dados Invalidos: numeros negativos.");
/* 11 */             }
                        ponta_de_prova(12);
/* 12 */                error_write(12,0,0,"#","#");
/* 12 */          }
                  ponta_de_prova(13);
/* 13 */          error_write(13,d,0,"!=","#");
                  fclose(path);
                  fclose(error);
/* 13 */       }
```

Programa 3:

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

FILE * error;
int nodecount;
char nom[10];
char ch1, ch2, ch3;

int en(char str1[], char str2[])  /* instrumentacao para o tratamento de strings */
{
 int i = 0;
 int err = 0;
 while  ( (str1[i] != '\0') || (str2[i] != '\0') )
 {
  err = err + (int) fabs( (int) (str1[i] - str2[i]) );
  i++;
 }
 return(err);
}

void error_write(int no, float exp1, float exp2, char *oprel, char *oplog)
 {
   fprintf(error,"%d %f %f %s %s\n", no, exp1, exp2, oprel, oplog);
   nodecount++;
   if (nodecount > 10000) exit(1);
 }

void char_test()
/*  1 */        {
                   FILE * path = fopen("path.tes","w");
                   static int printed_nodes = 0;
                   nodecount = 0;
                   error  = fopen("error.tes","w");
                   ponta_de_prova(1);
/*  1 */           error_write(1,0,0,"#","#");
/*  1 */           if(!strcmp(nom,"test"))
/*  2 */           {
                      ponta_de_prova(2);
/*  2 */              error_write(2,en(nom,"test"),0,"!=","#");
/*  2 */              if(ch1 == 'a')
/*  3 */              {
                         ponta_de_prova(3);
/*  3 */                 error_write(3,ch1,'a',"!=","#");
/*  3 */                 if(ch2 == 'b')
/*  4 */                 {
                            ponta_de_prova(4);
/*  4 */                    error_write(4,ch2,'b',"!=","#");
/*  4 */                    if (ch3 < 'c')
/*  5 */                    {
                               ponta_de_prova(5);
/*  5 */                       error_write(5,ch3,'c',">=","#");
/*  5 */                       printf("%s\n\n","*** Dados Encontrados ***");
                            }
                            ponta_de_prova(6);
```

```
/*  6 */                    error_write(6,ch3,'c',"<","#");
/*  6 */                 }
                         ponta_de_prova(7);
/*  7 */                 error_write(7,ch2,'b',"==","#");
/*  7 */              }
                      ponta_de_prova(8);
/*  8 */              error_write(8,ch1,'a',"==","#");
/*  8 */           }
                   ponta_de_prova(9);
/*  9 */           error_write(9,en(nom,"test"),0,"==","#");
                   fclose(path);
                   fclose(error);
/*  9 */        }

main()
{
 char aux;
 int i;
 i = 0;
 while ( (aux != ' ') && (i <= 3) )
   {
     scanf("%c\n",&aux);
     strcat(nom,&aux);
     i++;
   }
 scanf("%c\n",&ch1);
 scanf("%c\n",&ch2);
 scanf("%c\n",&ch3);
 char_test();
}
```

Programa 4:

```
#define ponta_de_prova(num) if(++printed_nodes % 10) fprintf(path," %2d ",num);\
else fprintf(path," %2d\n",num);

#include <stdio.h>
#include <string.h>
#include <stddef.h>
#include <math.h>

FILE * error;
int nodecount;
int a[257];

void error_write(int no, float exp1, float exp2, char *oprel, char *oplog)
 {
   fprintf(error,"%d %f %f %s %s\n", no, exp1, exp2, oprel, oplog);
   nodecount++;
   if (nodecount > 10000) exit(1);
 }

void find(int n,int f)
/*  1 */        {
                    FILE * path = fopen("path.tes","w");
                    static int printed_nodes = 0;
/*  1 */            int  b;
/*  1 */            int   m, ns, i, j, w;
                    error  = fopen("error.tes","w");
                    ponta_de_prova(1);
/*  1 */            b = 0;
/*  1 */            m = 1;
/*  1 */            ns = n;
/*  1 */            error_write(1,0,0,"#","#");
/*  2 */            while((m < ns) || b)
/*  3 */            {
                       ponta_de_prova(2);
/*  3 */               error_write(2,0,0,"#","#");
                       ponta_de_prova(3);
/*  3 */               error_write(3,m,ns,">=","&&");
/*  3 */               error_write(3,b,0,"NL","#");
/*  3 */               if(!b)
/*  4 */               {
                          ponta_de_prova(4);
/*  4 */                  error_write(4,!b,0,"NL","#");
/*  4 */                  i = m;
/*  4 */                  j = ns;
/*  4 */               }
/*  5 */               else
/*  5 */               {
                          ponta_de_prova(5);
/*  5 */                  error_write(5,!b,0,"PL","#");
/*  5 */                  b = 0;
                       }
                       ponta_de_prova(6);
/*  6 */               error_write(6,0,0,"#","#");
/*  6 */               if(i > j)
/*  7 */               {
                          ponta_de_prova(7);
/*  7 */                  error_write(7,i,j,"<=","#");
/*  7 */                  if(f > j)
/*  8 */                  {
                             ponta_de_prova(8);
```

```
/*  8 */                    error_write(8,f,j,"<=","#");
/*  8 */                    if(i > f)
/*  9 */                    {
                              ponta_de_prova(9);
/*  9 */                       error_write(9,i,f,"<=","#");
/*  9 */                       m = ns;
/*  9 */                    }
/* 10 */                    else
/* 10 */                    {
                              ponta_de_prova(10);
/* 10 */                       error_write(10,i,f,">","#");
/* 10 */                       m = i;
/* 10 */                    }
                           ponta_de_prova(11);
/* 11 */                    error_write(11,0,0,"#","#");
/* 11 */                 }
/* 12 */                 else
/* 12 */                 {
                           ponta_de_prova(12);
/* 12 */                    error_write(12,f,j,">","#");
/* 12 */                    ns = j;
/* 12 */                 }
                        ponta_de_prova(13);
/* 13 */                 error_write(13,0,0,"#","#");
/* 13 */              }
/* 14 */              else
/* 14 */              {
/* 14 */                 error_write(14,i,j,">","#");
/* 14 */                 while(a[i] < a[f])
/* 15 */                 {
                           ponta_de_prova(14);
/* 14 */                    error_write(14,0,0,"#","#");
                           ponta_de_prova(15);
/* 15 */                    error_write(15,a[i],a[f],">=","#");
/* 15 */                    i = i + 1;
/* 15 */                 }
                        ponta_de_prova(14);
/* 14 */                 error_write(14,0,0,"#","#");
/* 16 */                 error_write(16,a[i],a[f],"<","#");
/* 16 */                 while(a[f] < a[j])
/* 17 */                 {
                           ponta_de_prova(16);
/* 17 */                    error_write(16,0,0,"#","#");
                           ponta_de_prova(17);
/* 17 */                    error_write(17,a[f],a[j],">=","#");
/* 17 */                    j = j - 1 ;
/* 17 */                 }
                        ponta_de_prova(16);
/* 17 */                 error_write(16,0,0,"#","#");
                        ponta_de_prova(18);
/* 18 */                 error_write(18,a[f],a[j],"<","#");
/* 18 */                 if(i <= j)
/* 19 */                 {
                           ponta_de_prova(19);
/* 19 */                    error_write(19,i,j,">","#");
/* 19 */                    w = a[i];
/* 19 */                    a[i] = a[j];
/* 19 */                    a[j] = w;
/* 19 */                    i = i + 1;
/* 19 */                    j = j - 1;
/* 19 */                 }
                        ponta_de_prova(20);
```

```
/* 20 */                error_write(20,i,j,"<=","#");
/* 20 */                b = 1;
/* 20 */             }
                     ponta_de_prova(21);
/* 21 */             error_write(21,0,0,"#","#");
/* 21 */          }
                  ponta_de_prova(2);
/* 22 */          error_write(2,0,0,"#","#");
                  ponta_de_prova(22);
/* 22 */          error_write(22,m,ns,"<","||");
/* 22 */          error_write(22,b,0,"PL","#");
                  fclose(error);
                  fclose(path);
/* 22 */       }

main()
{
   char  *mystr;
   int  i;
   int N, F;
   char  *gets();
   void  find();
   mystr = (char *) malloc (80);
   nodecount = 0;
   scanf("%d", &N);
   scanf("%d", &F);
   N = fabs(N);
   F = fabs(F);
   for(i=1;i<=N;i++)
     scanf("%d", &a[i]);
   printf("%s\n","Entradas: ");
   printf("%s%d\n","N. Elementos: ",N);
   printf("%s%d\n","Pos. Pivo: ",F);
   for(i=1;i<=N;i++)
     printf("%s%d%s%d\n","elemento ",i," : ",a[i]);
   find(N, F);
   printf("%s\n","Saidas: ");
   printf("%s%d\n","N. Elementos: ",N);
   printf("%s%d\n","Pos. Pivo: ",F);
   for(i=1;i<=N;i++)
     printf("%s%d%s%d\n","elemento ",i," : ",a[i]);
}
```

Apêndice B

Programas Utilizados no Experimento

Neste apêndice encontram-se os programas utilizados no experimento descrito no Capítulo 6. São mostrados os grafos de fluxo de controle dos programas e os caminhos selecionados para o teste.

Programa 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

main()
/*  1 */        {
/*  1 */           float x, y, z, t, t2;
/*  1 */           printf("%s\n\n","entre com x, y e z ");
/*  1 */           scanf("%f\n",&x);
/*  1 */           scanf("%f\n",&y);
/*  1 */           scanf("%f\n",&z);
/*  1 */           printf("%s%f\n","X: ",x);
/*  1 */           printf("%s%f\n","Y: ",y);
/*  1 */           printf("%s%f\n\n","Z: ",z);
/*  1 */           if(z > y)
/*  2 */           {
/*  2 */              if(y > x)
/*  3 */              {
/*  3 */                 printf("%s\n","z > y > x ");
/*  3 */                 t = x + y;
/*  3 */                 if(t < z)
/*  4 */                 {
/*  4 */                    printf("%s\n","z > x + y");
/*  4 */                    t2 = x * y;
/*  4 */                    if( ((t2 - z) <= 5) && ((t2 - z) >= 0))
/*  5 */                    {
/*  5 */                       printf("%s\n","(x * y) - z <= 5 e (x * y) - z >= 0");
/*  5 */                    }
/*  6 */                 }
```

```
/*  7 */                else
/*  7 */                 {
/*  7 */                    printf("%s\n","z < = x + y");
/*  7 */                 }
/*  8 */              }
/*  9 */           }
/* 10 */        }
```

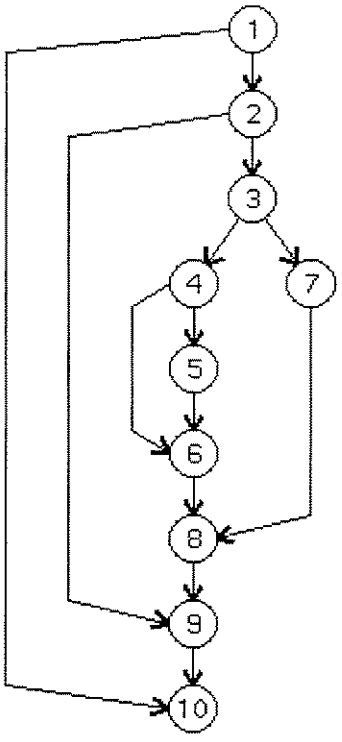

Figura B.1: Grafo do Programa 1

Caminhos Selecionados Para o Teste

1) 1 10

2) 1 2 9 10

3) 1 2 3 4 6 8 9 10

4) 1 2 3 4 5 6 8 9 10

5) 1 2 3 7 8 9 10

Programa 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <math.h>

main()
/*  1 */        {
/*  1 */           int n, d, q, r, t;
/*  1 */           printf("%s","Dividendo: ");
/*  1 */           scanf("%d\n",&n);
/*  1 */           printf("%s","Divisor: ");
/*  1 */           scanf("%d\n",&d);
/*  1 */           printf("\n");
/*  1 */           printf("%s%d\n","Dividendo: ",n);
/*  1 */           printf("%s%d\n","Divisor: ",d);
/*  1 */           printf("\n");
/*  1 */           if(d != 0)
/*  2 */            {
/*  2 */              if( (d > 0) && (n > 0) )
/*  3 */               {
/*  3 */                 q = 0;
/*  3 */                 r = n;
/*  3 */                 t = d;
/*  4 */                 while(r >= t)
/*  5 */                  {
/*  5 */                    t = t * 2;
/*  5 */                  }
/*  6 */                 while(t != d)
/*  7 */                  {
/*  7 */                    q = q * 2;
/*  7 */                    t = t / 2;
/*  7 */                    if(t <= r)
/*  8 */                     {
/*  8 */                       r = r - t;
/*  8 */                       q = q + 1;
/*  8 */                     }
/*  9 */                  }
/* 10 */                 printf("%s%d\n","Quociente: ",q);
/* 10 */                 printf("%s%d\n","Resto: ",r);
/* 10 */               }
/* 11 */              else
/* 11 */               {
/* 11 */                 printf("%s\n","Dados Invalidos: numeros negativos.");
/* 11 */               }
/* 12 */            }
/* 13 */        }
```

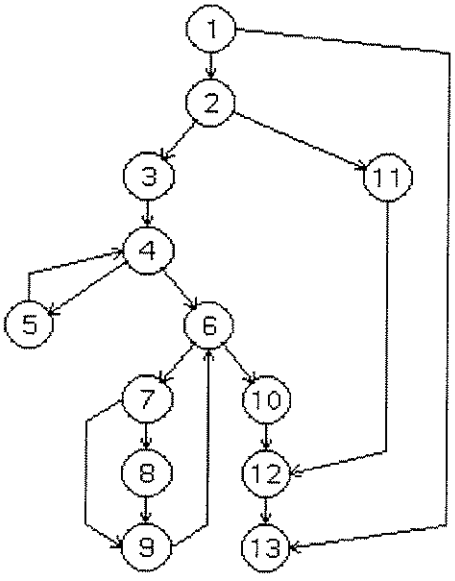

Figura B.2: Grafo do Programa 2

Caminhos Selecionados Para o Teste

1) 1 2 3 4 5 4 6 7 8 9 6 10 12 13

2) 1 2 3 4 5 4 5 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 8 9 6 7 9 6 7 9 6 7 8 9 6 7 8 9 6 10 12 13

3) 1 2 3 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 9 6 7 8 9 6 7 8 9 6 10 12 13

4) 1 2 3 4 5 4 6 7 8 9 6 10 12 13

5) 1 13

6) 1 2 11 12 13

7) 1 2 3 4 6 10 12 13

8) 1 2 3 4 5 4 5 4 5 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 8 9 6 7 8 9 6 7 8 9 6 7 9 6 7 9 6 7 9 6 10 12 13

9) 1 2 3 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 9 6 7 8 9 6 7 9 6 10 12 13

10) 1 2 3 4 5 4 5 4 6 7 8 9 6 7 9 6 10 12 13

Programa 3:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>

char nom[10];
char ch1, ch2, ch3;

void char_test()
/*  1 */        {
/*  1 */           if(!strcmp(nom,"test"))
/*  2 */           {
/*  2 */              if(ch1 == 'a')
/*  3 */              {
/*  3 */                 if(ch2 == 'b')
/*  4 */                 {
/*  4 */                    if (ch3 < 'c')
/*  5 */                    {
/*  5 */                       printf("%s\n\n","*** Dados Encontrados ***");
/*  5 */                    }
/*  6 */                 }
/*  7 */              }
/*  8 */           }
/*  9 */        }

main()
{
 char aux;
 int i;
 i = 0;
 while ( (aux != ' ') && (i <= 3) )
   {
     scanf("%c\n",&aux);
     strcat(nom,&aux);
     i++;
   }
 scanf("%c\n",&ch1);
 scanf("%c\n",&ch2);
 scanf("%c\n",&ch3);
 char_test();
}
```

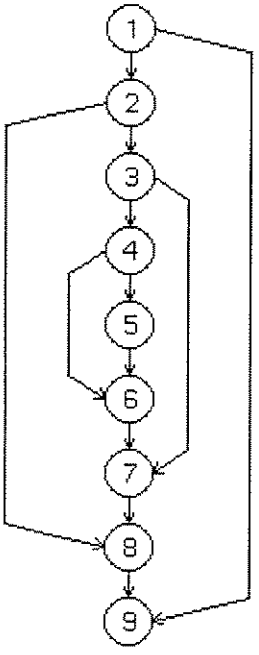

Figura B.3: Grafo do Programa 3

Caminhos Selecionados Para o Teste

1) 1 2 8 9

2) 1 2 3 7 8 9

3) 1 2 3 4 6 7 8 9

4) 1 2 3 4 5 6 7 8 9

Programa 4:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include <math.h>

int a [256+1];

void find(int n,int f)
/*  1 */        {
/*  1 */           int  b;
/*  1 */           int   m, ns, i, j, w;
/*  1 */           b = 0;
/*  1 */           m = 1;
/*  1 */           ns = n;
/*  2 */           while((m < ns) || b)
/*  3 */            {
/*  3 */              if(!b)
/*  4 */               {
/*  4 */                 i = m;
/*  4 */                 j = ns;
/*  4 */               }
/*  5 */              else
/*  5 */               {
/*  5 */                 b = 0;
/*  5 */               }
/*  6 */              if(i > j)
/*  7 */               {
/*  7 */                 if(f > j)
/*  8 */                  {
/*  8 */                    if(i > f)
/*  9 */                     {
/*  9 */                       m = ns;
/*  9 */                     }
/* 10 */                    else
/* 10 */                     {
/* 10 */                       m = i;
/* 10 */                     }
/* 11 */                  }
/* 12 */                 else
/* 12 */                  {
/* 12 */                    ns = j;
/* 12 */                  }
/* 13 */               }
/* 14 */              else
/* 14 */               {
/* 14 */                 while(a[i] < a[f])
/* 15 */                  {
/* 15 */                    i = i + 1;
/* 15 */                  }
/* 16 */                 while(a[f] < a[j])
/* 17 */                  {
/* 17 */                    j = j - 1 ;
/* 17 */                  }
/* 18 */                 if(i <= j)
/* 19 */                  {
/* 19 */                    w = a[i];
/* 19 */                    a[i] = a[j];
/* 19 */                    a[j] = w;
/* 19 */                    i = i + 1;
```

```
/* 19 */                    j = j - 1;
/* 19 */                  }
/* 20 */                 b = 1;
/* 20 */               }
/* 21 */            }
/* 22 */         }

main()
{
   char  *mystr;
   int  i;
   int N, F;
   char  *gets();
   void  find();

   mystr = (char *) malloc (80);
   scanf("%d", &N);
   scanf("%d", &F);
   N = fabs(N);
   F = fabs(F);
   for(i=1;i<=N;i++)
      {
       scanf("%d", &a[i]);
      }
   printf("%s\n","Entradas: ");
   printf("%s%d\n","N. Elementos: ",N);
   printf("%s%d\n","Pos. Pivo: ",F);
   for(i=1;i<=N;i++)
      {
       printf("%s%d%s%d\n","elemento ",i," : ",a[i]);
      }
   find(N, F);
   printf("%s\n","Saidas: ");
   printf("%s%d\n","N. Elementos: ",N);
   printf("%s%d\n","Pos. Pivo: ",F);
   for(i=1;i<=N;i++)
      {
      printf("%s%d%s%d\n","elemento ",i," : ",a[i]);
      }
}
```

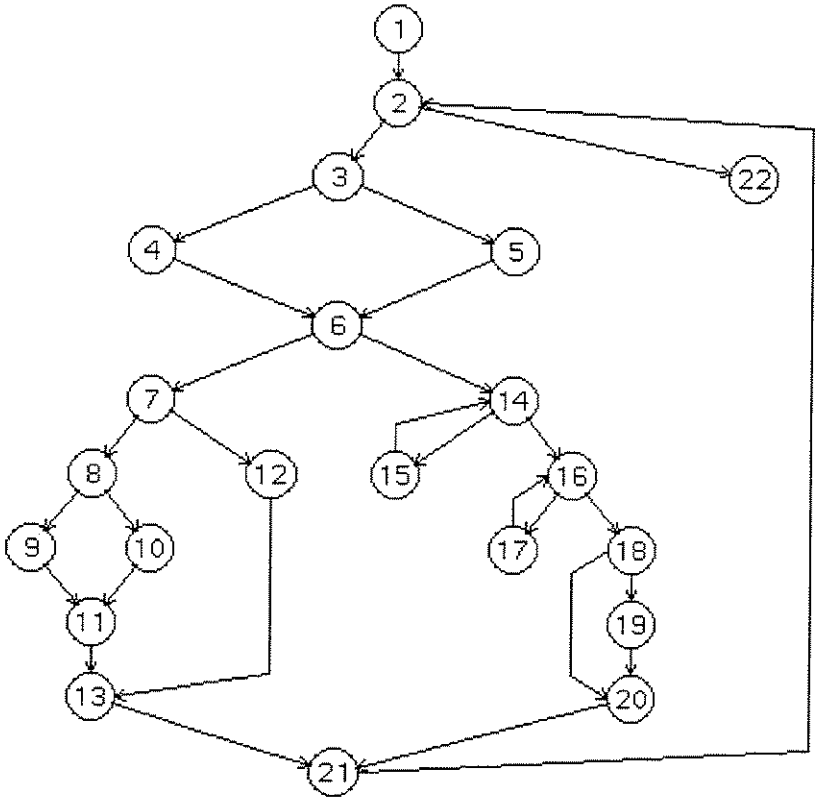

Figura B.4: Grafo do Programa 4

Caminhos Selecionados Para o Teste

1) 1 2 3 4 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 7 8 10 11 13 21 2 3 4 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 7 12 13 21 2 3 4 6 14 16 18 19 20 21 2 3 5 6 7 12 13 21 2 22

2) 1 2 3 4 6 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 16 18 19 20 21 2 3 5 6 7 8 9 11 13 21 2 22

3) 1 2 3 4 6 14 16 17 16 17 16 17 16 17 16 17 16 17 16 17 16 17 16 18 19 20 21 2 3 5 6 7 8 9 11 13 21 2 22

4) 1 2 3 4 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 14 16 18 19 20 21 2 3 5 6 7 12 13 21 2 3 4 6 14 15 14 15 14 15 14 15 14 15 14 16 18 19 20 21 2 3 5 6 7 8 9 11 13 21 2 22

5) 1 2 3 4 6 14 16 18 19 20 21 2 3 5 6 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 15 14 16 18 20 21 2 3 5 6 7 8 10 11 13 21 2 22

Apêndice C

Exemplo de Utilização da Ferramenta

Este apêndice apresenta um exemplo ilustrativo da utilização da ferramenta de geração automática de dados de teste. Foi utilizado o Programa 2, presente no Apêndice B, cuja versão instrumentada encontra-se no Apêndice A.

Foram selecionados para a geração de dados de teste os caminhos 2, 8 e 9 (vide Apêndice B e resultados do experimento no Capítulo 6, Seção 5.2.2). Foi também selecionado um caminho não executável, que permitirá ilustrar o tratamento feito pela ferramenta para este caso.

A seguir mostra-se o conteúdo do arquivo *campret.tes* que contém os caminhos selecionados para o teste:

```
1 2 3 4 5 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 8 9 6 7 9 6 7 9 6 7 8 9 6 7 8 9
 6 10 12 13 0

1 2 3 4 5 4 5 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 8 9 6 7 8 9 6 7 8 9 6 7 9 6
 7 9 6 7 9 6 10 12 13 0

1 2 3 4 5 4 5 4 5 4 5 4 6 7 8 9 6 7 9 6 7 8 9 6 7 9 6 10 12 13 0

1 2 3 4 6 7 8 9 6 7 9 6 7 8 9 6 7 9 6 10 12 13 0
```

Este arquivo servirá de entrada para ferramenta, portanto, tais caminhos serão tomados nesta ordem para a geração de dados de teste. Observe-se que os caminhos são terminados sempre pelo *flag* 0.

O quarto caminho do arquivo é não executável. Isto porque no nó 3 é feita a atribuição $t = d$. Como este caminho não passa no nó 5 (onde tem-se o comando $t = t \times 2$) a variável

t não é redefinida, fazendo com que o predicado $t \neq d$, referente ao arco $(6, 7)$, seja não factível. Deste modo o caminho completo como um todo é não executável.

O programa instrumentado foi compilado gerando o executável *testeprog2*.

A ferramenta foi executada, sendo fornecidos os seguintes parâmetros como entrada:

Parâmetros do Algoritmo Genético:

- Probabilidade de Recombinação $(Pr) = 0,8$;
- Probabilidade de Mutação $(Pm) = 0,02$;
- Tamanho da População $(Tp) = 40$;

Interface informada à ferramenta: variáveis de entrada $n1$ *e* $n2$, tipo "int", com ordem de grandeza = 250.

Em relação ao reuso de soluções passadas:

- Recuperação de soluções similares habilitada;
- Limite máximo de soluções recuperadas $= 0,5 \times Tp$;
- Nível mínimo de similaridade para recuperação de soluções = 3;
- Número máximo de casos inseridos na base para cada caminho pretendido = 15.000;

Observação: Por motivo de espaço as listagens foram resumidas, entretanto sem alterações de conteúdo. Nesses casos é utilizado o símbolo **(...)**.

```
DataGen - Iniciando Ferramenta

 ------------------------------------------------
|                                                |
|  ----- GERADOR DE DADOS DE TESTE  DataGen ----- |
|                                                |
|  Geracao de dados baseada em execucao dinamica; |
|  otimizacao utilizando Algoritmos Geneticos e  |
|  reuso de Casos Passados. - Prototipo 1.0      |
|________________________________________________|

ENTRADA DE DADOS:

Geracao Aleatoria (a) ou Baseada na Busca (b) ? (a|b): b

PARAMETROS DO ALGORITMO GENETICO:

Tamanho da populacao: 40
Probabilidade de recombinacao: 80
Probabilidade de mutacao: 2
periodicidade do relatorio (numero de geracoes): 2
Numero maximo de geracoes: 500

INFORMACOES SOBRE O PROGRAMA EM TESTE

Nome do arquivo que contem o executavel instrumentado: testeprog2
Nome do arquivo que contem o(s) caminho(s) pretendido(s): campret.tes
Numero de variaveis de entrada recebidas pelo teclado: 2

Descricao das variaveis de entrada recebidas pelo teclado - na ordem em que elas sao
recebidas pelo modulo

Variavel de entrada: 0
Nome da variavel: n1
Tipo da variavel (int | float | char | int[num] | char[num] | float[num]): int
Limite superior de valor da variavel (ordem de grandeza): 250

Variavel de entrada: 1
Nome da variavel: n2
Tipo da variavel (int | float | char | int[num] | char[num] | float[num]): int
Limite superior de valor da variavel (ordem de grandeza): 250

Numero de parametros recebidos por linha de comando: 0

INFORMACOES SOBRE O REUSO DE SOLUCOES PASSADAS

Recuperar dados que executaram caminhos similares para compor populacao inicial
de busca ? (s|n): s
Nivel minimo de similaridade para recuperacao (numero de nos coincidentes:
1 ... tam. caminho): 3
Eliminar informacoes de execucoes anteriores na base de solucoes ? (s|n): n
Numero maximo de caminhos a serem inseridos na base de solucoes durante a busca: 15000

DataGen - Reconhecendo tipos das variaveis de entrada
DataGen - Calculando informacoes para a codificacao das variaveis de entrada no cromossomo

Caminho Pretendido Numero 1:  1  2  3  4  5  4  5  4  5  4  5  4  5  4  5  4  6  7  8  9  6
  7  8  9  6  7  9  6  7  9  6  7  8  9  6  7  8  9  6  10  12  13

Enter para continuar

DataGen - Gerando Populacao Inicial do Algoritmo Genetico
Recuperado(s) 0 caminho(s) similar(es) para compor a populacao inicial.
```

```
Gerando Aleatoriamente Individuo Numero: 0
Gerando Aleatoriamente Individuo Numero: 1
Gerando Aleatoriamente Individuo Numero: 2
Gerando Aleatoriamente Individuo Numero: 3
(...)
Gerando Aleatoriamente Individuo Numero: 36
Gerando Aleatoriamente Individuo Numero: 37
Gerando Aleatoriamente Individuo Numero: 38
Gerando Aleatoriamente Individuo Numero: 39

Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: -168
Divisor: -62
Dados Invalidos: numeros negativos.
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 136
Divisor: 103
Quociente: 1
Resto: 33
```

```
 ------------------------------------------------
|                                                |
|  ----- GERADOR DE DADOS DE TESTE  DataGen ----- |
|                                                |
|  Geracao de dados baseada em execucao dinamica; |
|  otimizacao utilizando  Algoritmos Geneticos  e |
|  reuso  de  Casos  Passados.  -  Prototipo  1.0 |
|________________________________________________|
```

```
RELATORIO INICIAL

PARAMETROS DO ALGORITMO GENETICO:

Tamanho da Populacao: 40
Probabilidade de Recombinacao: 80
Probabilidade de Mutacao: 2

INFORMACOES SOBRE O PROGRAMA EM TESTE:

Nome do Executavel Instrumentado: testeprog2
Caminho Pretendido:  1  2  3  4  5  4  5  4  5  4  5  4  5  4  5  4  6
  7  8  9  6  7  8  9  6  7  9  6  7  9  6  7  8  9  6  7  8  9  6  10  12  13

Numero de variaveis de entrada recebidos pelo teclado: 2

Descricao das variaveis de entrada recebidas pelo teclado - na ordem em que elas sao
recebidas pelo modulo

Variavel de entrada: 0
Nome da variavel: n1
Tipo da variavel: int
Limite superior de valor da variavel: 250

Variavel de entrada: 1
Nome da variavel: n2
Tipo da variavel: int
Limite superior de valor da variavel: 250

Numero de parametros recebidos por linha de comando 0
```

```
INFORMACOES SOBRE O REUSO DE SOLUCOES:

Recuperar caminhos similares para compor populacao inicial de busca ?: s
Nivel minimo de similaridade para recuperacao (numero de nos coincidentes:
 1 ... tam. caminho): 3
Eliminar informacoes de execucoes anteriores na base de solucoes ?: n
Numero maximo de caminhos a serem inseridos na base de solucoes durante a busca: 15000

RELATORIO DA POPULACAO INICIAL: Geracao Numero: 0

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
--------------------------------------------------------------------------------------------
0 - 110101000100111110 0 - 0 - 0 - 2 - 11 - 1.486607 -
1 - 010101101010110100 0 - 0 - 0 - 4 - 6 - 3.961539 -
2 - 101111100010010101 0 - 0 - 0 - 2 - 11 - 1.723214 -
3 - 110000001001101101 0 - 0 - 0 - 2 - 11 - 1.712054 -
4 - 110011001001100011 0 - 0 - 0 - 2 - 11 - 1.658482 -
5 - 111000111111111001 0 - 0 - 0 - 2 - 11 - 1.000000 -
6 - 001110111011101111 0 - 0 - 0 - 4 - 6 - 3.340659 -
7 - 001110111101000001 0 - 0 - 0 - 2 - 11 - 1.854911 -
8 - 111011010011111010 0 - 0 - 0 - 2 - 11 - 1.513393 -
9 - 110011010110111011 0 - 0 - 0 - 2 - 11 - 1.238839 -
10 - 110001011000110001 0 - 0 - 0 - 2 - 11 - 1.689732 -
11 - 011110100010010100 0 - 0 - 0 - 6 - 6 - 5.257143 -
12 - 101101101110001110 0 - 0 - 0 - 2 - 11 - 1.439732 -
13 - 001000101000010000 0 - 0 - 0 - 10 - 6 - 9.000000 -
14 - 111111100001101001 0 - 0 - 0 - 2 - 11 - 1.437500 -
15 - 111000111110110001 0 - 0 - 0 - 2 - 11 - 1.160714 -
16 - 100010110100110101 0 - 0 - 0 - 2 - 11 - 1.832589 -
17 - 001001000011111110 0 - 0 - 0 - 4 - 6 - 3.000000 -
18 - 100000100000110110 0 - 0 - 0 - 2 - 11 - 1.991071 -
19 - 001110111000000011 0 - 0 - 0 - 22 - 9 - 21.000000 -
20 - 001011001111101001 0 - 0 - 0 - 2 - 11 - 1.479911 -
21 - 101100000111100010 0 - 0 - 0 - 2 - 11 - 1.281250 -
22 - 000011000010111001 0 - 0 - 0 - 4 - 6 - 3.115385 -
23 - 011001010101000111 0 - 0 - 0 - 2 - 11 - 1.841518 -
24 - 000001110001011101 0 - 0 - 0 - 4 - 6 - 3.565934 -
25 - 000100100100110110 0 - 0 - 0 - 2 - 11 - 1.879464 -
26 - 001000010100011111 0 - 0 - 0 - 2 - 11 - 1.930804 -
27 - 001001001100100000 0 - 0 - 0 - 2 - 11 - 1.928571 -
28 - 110100010101000100 0 - 0 - 0 - 2 - 11 - 1.486607 -
29 - 011001000111111111 0 - 0 - 0 - 2 - 11 - 1.430804 -
30 - 010001001100000011 0 - 0 - 0 - 2 - 11 - 1.993304 -
31 - 100000101011011000 0 - 0 - 0 - 2 - 11 - 1.988839 -
32 - 010110100100000110 0 - 0 - 0 - 2 - 11 - 1.986607 -
33 - 101100000000111101 0 - 0 - 0 - 2 - 11 - 1.785714 -
34 - 101100110011111000 0 - 0 - 0 - 2 - 11 - 1.772321 -
35 - 110111110100000100 0 - 0 - 0 - 2 - 11 - 1.566964 -
36 - 110100001100100010 0 - 0 - 0 - 2 - 11 - 1.564732 -
37 - 000100101110110011 0 - 0 - 0 - 2 - 11 - 1.600446 -
38 - 111011011101100110 0 - 0 - 0 - 2 - 11 - 1.283482 -
39 - 010001000001100111 0 - 0 - 0 - 6 - 6 - 5.000000 -
--------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 21.000000
Min. Ajuste: 1.000000
Media Ajuste: 2.694521
Soma Ajuste: 107.780838
Numero Recombinacoes.: 0
Numero Mutacoes: 0
```

```
Enter para continuar

GERACAO NUMERO: 1
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 119
Divisor: 3
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 123
Divisor: -65
Dados Invalidos: numeros negativos.

RELATORIO DE POPULACAO: Geracao Numero: 2

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 001110111000000011 0 - 0 - 0 - 22 - 9 - 21.000000 -
1 - 001000111000000010 19 - 13 - 6 - 22 - 9 - 21.000000 -
2 - 001010111000000011 0 - 5 - 17 - 14 - 6 - 13.000000 -
3 - 011110111000010001 0 - 5 - 17 - 12 - 6 - 11.791667 -
4 - 001000111000010011 19 - 14 - 16 - 8 - 6 - 7.946237 -
5 - 010101101000110000 19 - 14 - 16 - 8 - 6 - 7.795699 -
6 - 001001101000010000 36 - 5 - 18 - 10 - 6 - 9.732985 -
7 - 001110111000110101 36 - 5 - 18 - 8 - 6 - 7.000000 -
8 - 001000111000010000 19 - 7 - 18 - 10 - 6 - 9.701571 -
9 - 001111011101000011 19 - 7 - 18 - 2 - 11 - 1.736220 -
(...)
30 - 000011000010111100 13 - 8 - 15 - 4 - 6 - 3.088889 -
31 - 011001011000010001 13 - 8 - 15 - 12 - 6 - 11.425000 -
32 - 010000010101001100 2 - 24 - 18 - 2 - 11 - 1.700787 -
33 - 001001000011111100 2 - 24 - 18 - 4 - 6 - 3.000000 -
34 - 010001000000010000 27 - 8 - 7 - 12 - 6 - 11.000000 -
35 - 011001011100111001 27 - 8 - 7 - 2 - 11 - 1.775591 -
36 - 011110101000000000 18 - 33 - 16 - 1 - 13 - 0.000000 -
37 - 000011001000010000 18 - 33 - 16 - 6 - 6 - 5.872727 -
38 - 001110111000001001 7 - 0 - 4 - 12 - 6 - 11.791667 -
39 - 001111011101000001 7 - 0 - 4 - 2 - 11 - 1.744094 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 21.000000
Min. Ajuste: 0.000000
Media Ajuste: 7.112798
Soma Ajuste: 284.511932
Numero Recombinacoes.: 30
Numero Mutacoes: 47
Enter para continuar

GERACAO NUMERO: 3
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 119
Divisor: 3
Quociente: 39
Resto: 2
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 119
```

```
Divisor: 3
Quociente: 39
Resto: 2

RELATORIO DE POPULACAO: Geracao Numero: 4

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 001110111000000011 0 - 0 - 0 - 22 - 9 - 21.193548 -
1 - 001110111000000011 0 - 5 - 17 - 22 - 9 - 21.193548 -
2 - 001010111000010001 8 - 28 - 1 - 10 - 6 - 9.328767 -
3 - 011110111000001001 8 - 28 - 1 - 14 - 6 - 13.623854 -
4 - 001110101000000011 24 - 33 - 7 - 22 - 9 - 21.129032 -
5 - 001110111000011000 24 - 33 - 7 - 10 - 6 - 9.000000 -
6 - 001000111000010000 19 - 7 - 13 - 10 - 6 - 9.219178 -
7 - 011100101000010000 19 - 7 - 13 - 12 - 6 - 11.696630 -
8 - 001010011000001001 33 - 8 - 4 - 12 - 6 - 11.314607 -
9 - 001010111000100011 33 - 8 - 4 - 8 - 6 - 7.273973 -
(...)
30 - 001110011000000011 10 - 21 - 9 - 22 - 9 - 21.064516 -
31 - 001110001000000011 10 - 21 - 9 - 22 - 9 - 21.000000 -
32 - 011010111000010011 28 - 36 - 15 - 12 - 6 - 11.000000 -
33 - 001110111000000001 28 - 36 - 15 - 16 - 5 - 15.537815 -
34 - 000110011100000011 5 - 10 - 1 - 2 - 11 - 1.000000 -
35 - 000010111000000011 5 - 10 - 1 - 10 - 6 - 9.986301 -
36 - 001000111000001001 9 - 8 - 13 - 10 - 6 - 9.986301 -
37 - 001110111000000110 9 - 8 - 13 - 14 - 6 - 13.330276 -
38 - 001001011000001001 36 - 22 - 0 - 12 - 6 - 11.224719 -
39 - 001110111000000011 36 - 22 - 0 - 22 - 9 - 21.193548 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 25.000000
Min. Ajuste: 1.000000
Media Ajuste: 12.986814
Soma Ajuste: 519.472595
Numero Recombinacoes.: 64
Numero Mutacoes: 95

GERACAO NUMERO: 5
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 3
Divisor: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 119
Divisor: 2
Quociente: 59
Resto: 1

RELATORIO DE POPULACAO: Geracao Numero: 6

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 010101101000000011 0 - 37 - 10 - 26 - 8 - 25.444445 -
1 - 010101101000000011 0 - 37 - 10 - 26 - 8 - 25.444445 -
2 - 011110001000100111 35 - 0 - 17 - 10 - 6 - 9.000000 -
```

```
3 - 010101101000000001 35 - 0 - 17 - 16 - 5 - 15.000000 -
4 - 001100111000000011 28 - 7 - 10 - 22 - 9 - 21.000000 -
5 - 011001101001000011 28 - 7 - 10 - 8 - 6 - 7.000000 -
6 - 001110101010000111 26 - 35 - 13 - 4 - 6 - 3.735294 -
7 - 010110001000000011 26 - 35 - 13 - 26 - 8 - 25.000000 -
8 - 001110101000000011 26 - 1 - 8 - 22 - 9 - 21.341463 -
9 - 010101111000000011 26 - 1 - 8 - 26 - 8 - 25.222221 -
(...)
30 - 001110111000000010 32 - 2 - 14 - 26 - 8 - 25.222221 -
31 - 000000111000010011 32 - 2 - 14 - 4 - 6 - 3.823529 -
32 - 001011101001000011 26 - 11 - 7 - 6 - 6 - 5.655462 -
33 - 010110001000000111 26 - 11 - 7 - 14 - 6 - 13.000000 -
34 - 001010011010010111 38 - 28 - 16 - 4 - 6 - 3.000000 -
35 - 001100111000000011 38 - 28 - 16 - 22 - 9 - 21.000000 -
36 - 001110001000000011 14 - 13 - 16 - 22 - 9 - 21.243902 -
37 - 011001111000000011 14 - 13 - 16 - 16 - 5 - 15.862386 -
38 - 000001111000000001 1 - 36 - 17 - 12 - 6 - 11.987342 -
39 - 001110111000000010 1 - 36 - 17 - 26 - 8 - 25.222221 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 25.444445
Min. Ajuste: 0.000000
Media Ajuste: 14.241551
Soma Ajuste: 569.662048
Numero Recombinacoes.: 97
Numero Mutacoes: 149
Enter para continuar

GERACAO NUMERO: 7
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 3
Quociente: 57
Resto: 2
Avaliando Individuo Numero: 1
Dividendo: Divisor:
Dividendo: 173
Divisor: 3
Quociente: 57
Resto: 2
Avaliando Individuo Numero: 2
Dividendo: Divisor:
Dividendo: 102
Divisor: -3
Dados Invalidos: numeros negativos.
Avaliando Individuo Numero: 3
Dividendo: Divisor:
Dividendo: 173
Divisor: 3
Quociente: 57
Resto: 2
Avaliando Individuo Numero: 4
Dividendo: Divisor:
Dividendo: 103
Divisor: 2
Quociente: 51
Resto: 1

DataGen: ******** SOLUCAO ENCONTRADA ********
```

```
Caminho Pretendido:  1  2  3  4  5  4  5  4  5  4  5  4  5  4  5  4  6  7  8  9  6
 7  8  9  6  7  9  6  7  9  6  7  8  9  6  7  8  9  6  10  12  13

Numero da Geracao: 7
Numero da Tentativa: 285

N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross.. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
4 001100111000000O10 35 30 16 42 0 42

Dados de entrada da solucao encontram-se no arquivo <dadosent.tes>
Parametros passados ao programa encontram-se no arquivo <execprog>

Enter para continuar

DataGen - Iniciando Ferramenta

Caminho Pretendido Numero 2:  1  2  3  4  5  4  5  4  5  4  5  4  5  4  5  4  5  4  5  4
  6  7  8  9  6  7  8  9  6  7  8  9  6  7  8  9  6  7  9  6  7  9  6  7  9  6  10  12  13

Enter para continuar

DataGen - Gerando Populacao Inicial do Algoritmo Genetico
Datagen - Recuperando Caminhos com Similaridade: 47
Datagen - Recuperando Caminhos com Similaridade: 46
Datagen - Recuperando Caminhos com Similaridade: 45
(...)
Datagen - Recuperando Caminhos com Similaridade: 19
Datagen - Recuperando Caminhos com Similaridade: 18
Datagen - Recuperando Caminhos com Similaridade: 17
Recuperado(s) 20 caminho(s) similar(es) para compor a populacao inicial.
Gerando Aleatoriamente Individuo Numero: 20
Gerando Aleatoriamente Individuo Numero: 21
Gerando Aleatoriamente Individuo Numero: 22
(...)
Gerando Aleatoriamente Individuo Numero: 37
Gerando Aleatoriamente Individuo Numero: 38
Gerando Aleatoriamente Individuo Numero: 39

Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 247
Divisor: 2
Quociente: 123
Resto: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 171
Divisor: -97
Dados Invalidos: numeros negativos.

RELATORIO DA POPULACAO INICIAL: Geracao Numero: 0

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 011110111000000010 0 - 37 - 10 - 39 - 8 - 38.000000 -
1 - 011100101000000010 0 - 37 - 10 - 32 - 9 - 31.000000 -
2 - 001110111000000001 35 - 0 - 12 - 32 - 9 - 31.909090 -
3 - 011110110000000011 35 - 0 - 12 - 24 - 9 - 23.481482 -
4 - 011110111000000011 35 - 30 - 16 - 24 - 9 - 23.493828 -
```

```
5 - 011111111000000011 35 - 30 - 16 - 24 - 9 - 23.592592 -
6 - 011101101000000011 20 - 4 - 13 - 24 - 9 - 23.370371 -
7 - 010011111000000010 20 - 4 - 13 - 24 - 9 - 23.592592 -
8 - 011001111000000011 7 - 33 - 0 - 24 - 9 - 23.000000 -
9 - 010101101000000001 7 - 33 - 0 - 18 - 5 - 17.000000 -
10 - 001110111000000011 29 - 30 - 7 - 16 - 6 - 15.051949 -
11 - 001110111000000011 29 - 30 - 7 - 16 - 6 - 15.051949 -
12 - 001000111000000010 24 - 20 - 0 - 16 - 6 - 15.259740 -
13 - 001110111000000011 24 - 20 - 0 - 16 - 6 - 15.051949 -
14 - 001000111000000010 21 - 8 - 18 - 16 - 6 - 15.259740 -
15 - 001110111000000011 21 - 8 - 18 - 16 - 6 - 15.051949 -
16 - 001110111000000011 1 - 2 - 4 - 16 - 6 - 15.051949 -
17 - 001110111000000011 1 - 2 - 4 - 16 - 6 - 15.051949 -
18 - 001110011000000011 38 - 22 - 11 - 16 - 6 - 15.000000 -
19 - 001000111000000010 38 - 22 - 11 - 16 - 6 - 15.259740 -
20 - 000001001100011111 1 - 11 - 14 - 2 - 11 - 1.916216 -
21 - 110101110001000000 1 - 11 - 14 - 2 - 11 - 1.529730 -
22 - 110100000101001111 26 - 16 - 5 - 2 - 11 - 1.354054 -
23 - 010100001010100110 26 - 16 - 5 - 4 - 6 - 3.943182 -
24 - 111000011110101111 22 - 23 - 13 - 2 - 11 - 1.000000 -
25 - 111111010001001000 22 - 23 - 13 - 2 - 11 - 1.324324 -
26 - 110110010110001000 9 - 4 - 0 - 2 - 11 - 1.151351 -
27 - 010101100000010110 9 - 4 - 0 - 10 - 6 - 9.000000 -
28 - 101011001101010110 7 - 25 - 17 - 2 - 11 - 1.527027 -
29 - 000100111101111101 7 - 25 - 17 - 2 - 11 - 1.662162 -
30 - 001101100101111111 39 - 23 - 10 - 2 - 11 - 1.656757 -
31 - 001101101001011111 39 - 23 - 10 - 6 - 6 - 5.000000 -
32 - 110101111100000100 30 - 22 - 6 - 2 - 11 - 1.516216 -
33 - 000101111010000111 30 - 22 - 6 - 4 - 6 - 3.000000 -
34 - 010101000110001100 0 - 23 - 6 - 2 - 11 - 1.621622 -
35 - 000000110110001010 0 - 23 - 6 - 2 - 11 - 1.627027 -
36 - 010101010111100101 3 - 30 - 18 - 2 - 11 - 1.381081 -
37 - 000010011110101101 3 - 30 - 18 - 2 - 11 - 1.532432 -
38 - 111000001101110011 1 - 7 - 9 - 2 - 11 - 1.167567 -
39 - 010101011101100001 1 - 7 - 9 - 2 - 11 - 1.737838 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 38.000000
Min. Ajuste: 1.000000
Media Ajuste: 11.354484
Soma Ajuste: 454.179352
Numero Recombinacoes.: 0
Numero Mutacoes: 0
Enter para continuar

GERACAO NUMERO: 1
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 247
Divisor: 2
Quociente: 123
Resto: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 191
Divisor: 2
Quociente: 95
Resto: 1

RELATORIO DE POPULACAO: Geracao Numero: 2
```

```
Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 011110111000000010 0 - 37 - 10 - 39 - 8 - 38.000000 -
1 - 011110111000000010 0 - 0 - 11 - 39 - 8 - 38.000000 -
2 - 011010111000000010 17 - 0 - 1 - 28 - 9 - 27.500000 -
3 - 110011111001000011 17 - 0 - 1 - 2 - 11 - 1.353659 -
4 - 011110111000000000 20 - 10 - 1 - 1 - 13 - 0.000000 -
5 - 111110110000001010 20 - 10 - 1 - 2 - 11 - 1.000000 -
6 - 011001110000000011 9 - 20 - 18 - 24 - 9 - 23.000000 -
7 - 011110110000000011 9 - 20 - 18 - 24 - 9 - 23.487804 -
8 - 011110111000000111 10 - 19 - 9 - 16 - 6 - 15.000000 -
9 - 011101101000000010 10 - 19 - 9 - 32 - 9 - 31.666666 -
(...)
30 - 010101110000000010 15 - 11 - 11 - 24 - 9 - 23.780487 -
31 - 001000111000000011 15 - 11 - 11 - 14 - 6 - 13.000000 -
32 - 011001110000000010 9 - 29 - 9 - 28 - 9 - 27.000000 -
33 - 010011111000000011 9 - 29 - 9 - 16 - 6 - 15.835821 -
34 - 001110111000000011 27 - 30 - 0 - 16 - 6 - 15.636816 -
35 - 011010111000000010 27 - 30 - 0 - 28 - 9 - 27.500000 -
36 - 011100111000000010 4 - 5 - 5 - 32 - 9 - 31.000000 -
37 - 001110110100000011 4 - 5 - 5 - 2 - 11 - 1.987805 -
38 - 001000101000000010 38 - 3 - 8 - 16 - 6 - 15.706468 -
39 - 010111111000000010 38 - 3 - 8 - 24 - 9 - 23.987804 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 38.000000
Min. Ajuste: 0.000000
Media Ajuste: 17.633745
Soma Ajuste: 705.349792
Numero Recombinacoes.: 32
Numero Mutacoes: 42
Enter para continuar

GERACAO NUMERO: 3
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 247
Divisor: 2
Quociente: 123
Resto: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 157
Divisor: 2
Quociente: 78
Resto: 1

RELATORIO DE POPULACAO: Geracao Numero: 4

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 011110111000000010 0 - 37 - 10 - 39 - 8 - 38.000000 -
1 - 011110111000000010 1 - 7 - 18 - 39 - 8 - 38.000000 -
2 - 011100111000000010 33 - 0 - 8 - 32 - 9 - 31.000000 -
3 - 011110111000000010 33 - 0 - 8 - 39 - 8 - 38.000000 -
4 - 011011111000000010 3 - 12 - 14 - 28 - 9 - 27.799999 -
5 - 010110111000000010 3 - 12 - 14 - 24 - 9 - 23.742857 -
```

```
6 - 010101110001000010 38 - 11 - 13 - 8 - 6 - 7.000000 -
7 - 010011111000100010 38 - 11 - 13 - 10 - 6 - 9.000000 -
8 - 111100111000000010 33 - 6 - 9 - 2 - 11 - 1.000000 -
9 - 011011011000000010 33 - 6 - 9 - 28 - 9 - 27.000000 -
(...)
30 - 011110111000000010 0 - 3 - 18 - 39 - 8 - 38.000000 -
31 - 011011111000010010 0 - 3 - 18 - 12 - 6 - 11.000000 -
32 - 011101101000100010 9 - 1 - 10 - 10 - 6 - 9.690266 -
33 - 011110111000100010 9 - 1 - 10 - 10 - 6 - 9.778761 -
34 - 001000111001000011 13 - 24 - 18 - 6 - 6 - 5.086957 -
35 - 000011111000000010 13 - 24 - 18 - 12 - 6 - 11.984615 -
36 - 010111111010000010 10 - 6 - 12 - 6 - 6 - 5.000000 -
37 - 011111111000000010 10 - 6 - 12 - 36 - 8 - 35.000000 -
38 - 011101101000000010 35 - 8 - 18 - 32 - 9 - 31.666666 -
39 - 010011101000000010 35 - 8 - 18 - 24 - 9 - 23.000000 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 38.000000
Min. Ajuste: 0.000000
Media Ajuste: 20.048883
Soma Ajuste: 801.955322
Numero Recombinacoes.: 58
Numero Mutacoes: 78
Enter para continuar

GERACAO NUMERO: 5
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 247
Divisor: 2
Quociente: 123
Resto: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 245
Divisor: 18
Quociente: 13
Resto: 11

RELATORIO DE POPULACAO: Geracao Numero: 12

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 011110101000000010 1 - 1 - 7 - 39 - 8 - 38.666668 -
1 - 011110101000000010 1 - 13 - 18 - 39 - 8 - 38.666668 -
2 - 011100101000000010 39 - 1 - 0 - 32 - 9 - 31.000000 -
3 - 011110111000000010 39 - 1 - 0 - 39 - 8 - 38.000000 -
4 - 011110111000000010 2 - 38 - 9 - 39 - 8 - 38.000000 -
5 - 011110101000000110 2 - 38 - 9 - 16 - 6 - 15.000000 -
6 - 011110111000001011 31 - 10 - 17 - 14 - 6 - 13.000000 -
7 - 011101101000000010 31 - 10 - 17 - 32 - 9 - 31.727272 -
8 - 011000111000000010 34 - 0 - 14 - 28 - 9 - 27.000000 -
9 - 001110101000000010 34 - 0 - 14 - 16 - 6 - 15.920863 -
(...)
30 - 011100101000100010 38 - 18 - 7 - 10 - 6 - 9.000000 -
31 - 011110111000000010 38 - 18 - 7 - 39 - 8 - 38.000000 -
32 - 011110111000000010 39 - 1 - 15 - 39 - 8 - 38.000000 -
33 - 011110101000000010 39 - 1 - 15 - 39 - 8 - 38.666668 -
34 - 011010101000000011 23 - 11 - 18 - 24 - 9 - 23.000000 -
```

```
35 - 111110111000000010 23 - 11 - 18 - 2 - 11 - 1.000000 -
36 - 011110111000000010 18 - 8 - 7 - 39 - 8 - 38.000000 -
37 - 011100111000000010 18 - 8 - 7 - 32 - 9 - 31.181818 -
38 - 011101111000000011 1 - 10 - 0 - 24 - 9 - 23.346666 -
39 - 011110101000010010 1 - 10 - 0 - 12 - 6 - 11.122449 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 38.666668
Min. Ajuste: 1.000000
Media Ajuste: 25.982462
Soma Ajuste: 1039.298462
Numero Recombinacoes.: 175
Numero Mutacoes: 256
Enter para continuar

GERACAO NUMERO: 13
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 245
Divisor: 2
Quociente: 122
Resto: 1
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 237
Divisor: 2
Quociente: 118
Resto: 1

RELATORIO DE POPULACAO: Geracao Numero: 14

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 011110101000000010 1 - 1 - 7 - 39 - 8 - 38.666668 -
1 - 011110101000000010 22 - 3 - 5 - 39 - 8 - 38.666668 -
2 - 011101111000000010 9 - 30 - 6 - 32 - 9 - 31.916666 -
3 - 011110101000000010 9 - 30 - 6 - 39 - 8 - 38.666668 -
4 - 011100100000000010 28 - 36 - 9 - 32 - 9 - 31.000000 -
5 - 111100110000000010 28 - 36 - 9 - 2 - 11 - 1.000000 -
6 - 010100101000000010 19 - 22 - 13 - 24 - 9 - 23.341463 -
7 - 011010101001000011 19 - 22 - 13 - 8 - 6 - 7.000000 -
8 - 011110101000000010 2 - 0 - 18 - 39 - 8 - 38.666668 -
9 - 010110101000000010 2 - 0 - 18 - 24 - 9 - 23.731707 -
(...)
30 - 011101101000000010 29 - 1 - 18 - 32 - 9 - 31.750000 -
31 - 011110101000000010 29 - 1 - 18 - 39 - 8 - 38.666668 -
32 - 011110111000000010 6 - 31 - 6 - 39 - 8 - 38.000000 -
33 - 011110111000100010 6 - 31 - 6 - 10 - 6 - 9.000000 -
34 - 001101101000000111 19 - 24 - 18 - 12 - 6 - 11.000000 -
35 - 001110101000000010 19 - 24 - 18 - 16 - 6 - 15.928104 -
36 - 011110111000000011 39 - 23 - 11 - 24 - 9 - 23.000000 -
37 - 011110111000000000 39 - 23 - 11 - 1 - 13 - 0.000000 -
38 - 011110111000000010 33 - 29 - 11 - 39 - 8 - 38.000000 -
39 - 011101101000000010 33 - 29 - 11 - 32 - 9 - 31.750000 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 38.666668
Min. Ajuste: 0.000000
```

```
Media Ajuste: 26.502359
Soma Ajuste: 1060.094360
Numero Recombinacoes.: 204
Numero Mutacoes: 293
Enter para continuar

GERACAO NUMERO: 15
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 245
Divisor: 2
Quociente: 122
Resto: 1
(...)
Avaliando Individuo Numero: 17
Dividendo: Divisor:
Dividendo: 229
Divisor: 2
Quociente: 114
Resto: 1
Avaliando Individuo Numero: 18
Dividendo: Divisor:
Dividendo: 247
Divisor: 3
Quociente: 82
Resto: 1
Avaliando Individuo Numero: 19
Dividendo: Divisor:
Dividendo: 241
Divisor: 2
Quociente: 120
Resto: 1
DataGen: ******** SOLUCAO ENCONTRADA ********

Caminho Pretendido:  1  2  3  4  5  4  5  4  5  4  5  4  5  4  5  4  5  4  6  7  8  9  6
  7  8  9  6  7  8  9  6  7  8  9  6  7  9  6  7  9  6  7  9  6  10  12  13

Numero da Geracao: 15
Numero da Tentativa: 620

N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross.. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
19 011110001000000010 10 16 0 47 0 47

Dados de entrada da solucao encontram-se no arquivo <dadosent.tes>
Parametros passados ao programa encontram-se no arquivo <execprog>

Enter para continuar

DataGen - Iniciando Ferramenta

Caminho Pretendido Numero 3:  1  2  3  4  5  4  5  4  5  4  5  4  6  7  8  9  6  7  9  6  7
  8  9  6  7  9  6  10  12  13

Enter para continuar

DataGen - Gerando Populacao Inicial do Algoritmo Genetico
Datagen - Recuperando Caminhos com Similaridade: 30
DataGen - DADO DE ENTRADA ENCONTRADO NA BASE DE SOLUCOES
Recuperado(s) 1 caminho(s) similar(es) para compor a populacao inicial.
Gerando Aleatoriamente Individuo Numero: 1
Gerando Aleatoriamente Individuo Numero: 2
```

```
(...)
Gerando Aleatoriamente Individuo Numero: 38
Gerando Aleatoriamente Individuo Numero: 39

Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 113
Divisor: 11
Quociente: 10
Resto: 3

DataGen: ******** SOLUCAO ENCONTRADA ********

Caminho Pretendido:  1  2  3  4  5  4  5  4  5  4  5  4  6  7  8  9  6  7  9  6  7  8  9
  6  7  9  6  10  12  13

Numero da Geracao: 0
Numero da Tentativa: 1

N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross.. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 001110001000001O11  0 0 0 30 0 30

Dados de entrada da solucao encontram-se no arquivo <dadosent.tes>
Parametros passados ao programa encontram-se no arquivo <execprog>

Caminho Pretendido Numero 4:  1  2  3  4  6  7  8  9  6  7  9  6  7  8  9  6  7  9  6  10  12  13

Enter para continuar

DataGen - Gerando Populacao Inicial do Algoritmo Genetico
Datagen - Recuperando Caminhos com Similaridade: 22
Datagen - Recuperando Caminhos com Similaridade: 21
Datagen - Recuperando Caminhos com Similaridade: 20
(...)
Datagen - Recuperando Caminhos com Similaridade: 8
Datagen - Recuperando Caminhos com Similaridade: 7
Datagen - Recuperando Caminhos com Similaridade: 6
Recuperado(s) 20 caminho(s) similar(es) para compor a populacao inicial.
Gerando Aleatoriamente Individuo Numero: 20
Gerando Aleatoriamente Individuo Numero: 21
Gerando Aleatoriamente Individuo Numero: 22
(...)
Gerando Aleatoriamente Individuo Numero: 37
Gerando Aleatoriamente Individuo Numero: 38
Gerando Aleatoriamente Individuo Numero: 39

Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: -238
Divisor: 107
Dados Invalidos: numeros negativos.

RELATORIO DA POPULACAO INICIAL: Geracao Numero: 0
```

```
Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 0101011010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 0011101111011101111 0 - 0 - 0 - 5 - 10 - 5.000000 -
2 - 0010010000111111110 0 - 0 - 0 - 5 - 10 - 5.000000 -
3 - 0000110000101111001 0 - 0 - 0 - 5 - 10 - 5.000000 -
4 - 0000011100010111101 0 - 0 - 0 - 5 - 10 - 5.000000 -
5 - 0000110000101100001 0 - 0 - 0 - 5 - 10 - 5.000000 -
6 - 0010010000111111001 0 - 0 - 0 - 5 - 10 - 5.000000 -
7 - 0000110000101111111 0 - 0 - 0 - 5 - 10 - 5.000000 -
8 - 0010010000111111100 0 - 0 - 0 - 5 - 10 - 5.000000 -
9 - 0101011010110110110 0 - 0 - 0 - 5 - 10 - 5.000000 -
10 - 0010000100010111001 0 - 0 - 0 - 5 - 10 - 5.000000 -
11 - 0110010000111111110 0 - 0 - 0 - 5 - 10 - 5.000000 -
12 - 0010010000111111100 0 - 0 - 0 - 5 - 10 - 5.000000 -
13 - 0101011000111111001 0 - 0 - 0 - 5 - 10 - 5.000000 -
14 - 0010001010110110001 0 - 0 - 0 - 5 - 10 - 5.000000 -
15 - 0000011010000010011 0 - 0 - 0 - 5 - 10 - 5.000000 -
16 - 0010011000101111001 0 - 0 - 0 - 5 - 10 - 5.000000 -
17 - 0000110000101111100 0 - 0 - 0 - 5 - 10 - 5.000000 -
18 - 0010010000111111100 0 - 0 - 0 - 5 - 10 - 5.000000 -
19 - 0010001000011111110 0 - 0 - 0 - 5 - 10 - 5.000000 -
20 - 0100111011100011011 0 - 0 - 0 - 2 - 11 - 1.593176 -
21 - 1100100011101011101 0 - 0 - 0 - 2 - 11 - 1.165354 -
22 - 1001011011101011100 0 - 0 - 0 - 2 - 11 - 1.430446 -
23 - 0100110110010101111 0 - 0 - 0 - 4 - 5 - 3.000000 -
24 - 1110011110111100010 0 - 0 - 0 - 2 - 11 - 1.456693 -
25 - 0101100110011011111 0 - 0 - 0 - 4 - 5 - 3.000000 -
26 - 0001001101101001000 0 - 0 - 0 - 2 - 11 - 1.569554 -
27 - 1111010111100001111 0 - 0 - 0 - 2 - 11 - 1.028871 -
28 - 1010111011111011110 0 - 0 - 0 - 2 - 11 - 1.173228 -
29 - 1000010010011101101 0 - 0 - 0 - 2 - 11 - 1.976378 -
30 - 0010100000111011101 0 - 0 - 0 - 5 - 10 - 5.000000 -
31 - 1110100001101011101 0 - 0 - 0 - 2 - 11 - 1.000000 -
32 - 1111110110101011100 0 - 0 - 0 - 2 - 11 - 1.341207 -
33 - 0011000000011111110 0 - 0 - 0 - 5 - 10 - 5.000000 -
34 - 0101000101110111101 0 - 0 - 0 - 2 - 11 - 1.419948 -
35 - 0001100001001000001 0 - 0 - 0 - 2 - 11 - 1.913386 -
36 - 1100111101000111101 0 - 0 - 0 - 2 - 11 - 1.509186 -
37 - 1100111111010011100 0 - 0 - 0 - 2 - 11 - 1.582677 -
38 - 0010101001011100000 0 - 0 - 0 - 2 - 11 - 1.706037 -
39 - 1111011100011011011 0 - 0 - 0 - 2 - 11 - 1.375328 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 3.481037
Soma Ajuste: 139.241486
Numero Recombinacoes.: 0
Numero Mutacoes: 0

GERACAO NUMERO: 1
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
```

```
Dividendo: Divisor:
Dividendo: 14
Divisor: 185
Quociente: 0
Resto: 14

RELATORIO DE POPULACAO: Geracao Numero: 4

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 010101101010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 010101001010110001 14 - 35 - 18 - 5 - 10 - 5.000000 -
2 - 000011000011111000 7 - 17 - 18 - 5 - 10 - 5.000000 -
3 - 000001110001111000 7 - 17 - 18 - 5 - 10 - 5.000000 -
4 - 010101101010111111 11 - 22 - 10 - 5 - 10 - 4.000000 -
5 - 010001101010110010 11 - 22 - 10 - 5 - 10 - 5.000000 -
6 - 010001101011111100 3 - 23 - 18 - 5 - 10 - 5.000000 -
7 - 000001010010110000 3 - 23 - 18 - 5 - 10 - 5.000000 -
8 - 001001101010110000 10 - 38 - 18 - 5 - 10 - 4.265403 -
9 - 010101001010110001 10 - 38 - 18 - 5 - 10 - 4.260664 -
(...)
30 - 010001000001011000 31 - 11 - 16 - 4 - 5 - 3.428571 -
31 - 010101101010110011 31 - 11 - 16 - 5 - 10 - 4.938389 -
32 - 010101000010111110 26 - 36 - 11 - 5 - 10 - 4.815166 -
33 - 010110011001111100 26 - 36 - 11 - 4 - 5 - 3.345238 -
34 - 000001110010111001 32 - 13 - 3 - 5 - 10 - 4.928910 -
35 - 000011000010111100 32 - 13 - 3 - 5 - 10 - 5.000000 -
36 - 000011110010110000 33 - 39 - 9 - 5 - 10 - 4.748815 -
37 - 010101101001011001 33 - 39 - 9 - 4 - 5 - 3.000000 -
38 - 001001000011111111 29 - 13 - 18 - 5 - 10 - 5.000000 -
39 - 000001110010111001 29 - 13 - 18 - 5 - 10 - 4.677725 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 4.537969
Soma Ajuste: 181.518768
Numero Recombinacoes.: 52
Numero Mutacoes: 59
Enter para continuar

GERACAO NUMERO: 5
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173

RELATORIO DE POPULACAO: Geracao Numero: 10

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
```

```
-------------------------------------------------------------------------------------------
0 - 010101101010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 010100010011111000 19 - 34 - 13 - 5 - 10 - 5.000000 -
2 - 000001110001101000 2 - 25 - 18 - 5 - 10 - 4.521327 -
3 - 010101101011110101 2 - 25 - 18 - 5 - 10 - 5.000000 -
4 - 001010001011111000 38 - 35 - 18 - 5 - 10 - 4.000000 -
5 - 010100010011111000 38 - 35 - 18 - 5 - 10 - 4.781991 -
6 - 110101101011100001 10 - 21 - 9 - 2 - 11 - 1.199074 -
7 - 001001110010110100 10 - 21 - 9 - 5 - 10 - 5.000000 -
8 - 001001000010110101 15 - 24 - 9 - 5 - 10 - 4.265403 -
9 - 000100101010010001 15 - 24 - 9 - 5 - 10 - 4.819905 -
(...)
30 - 010101100010010000 27 - 20 - 6 - 4 - 5 - 3.000000 -
31 - 000001000011111000 27 - 20 - 6 - 5 - 10 - 4.658768 -
32 - 001001000010011100 34 - 22 - 18 - 5 - 10 - 4.872038 -
33 - 001001010010101101 34 - 22 - 18 - 5 - 10 - 4.739336 -
34 - 010101101110110100 18 - 39 - 18 - 2 - 11 - 1.166667 -
35 - 000111001010111000 18 - 39 - 18 - 5 - 10 - 5.000000 -
36 - 000111001011111010 9 - 1 - 15 - 5 - 10 - 4.331754 -
37 - 001001101011111100 9 - 1 - 15 - 5 - 10 - 4.729858 -
38 - 001111000011111000 16 - 0 - 18 - 5 - 10 - 4.786730 -
39 - 010101101010110100 16 - 0 - 18 - 5 - 10 - 4.677725 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 4.373755
Soma Ajuste: 174.950211
Numero Recombinacoes.: 131
Numero Mutacoes: 141
Enter para continuar

GERACAO NUMERO: 11
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 223
Divisor: 149
Quociente: 1
Resto: 74

RELATORIO DE POPULACAO: Geracao Numero: 70

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 010101101010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 001011000010100001 16 - 38 - 3 - 5 - 10 - 5.000000 -
2 - 001011000110100001 32 - 35 - 15 - 2 - 11 - 1.018293 -
3 - 000010011010100001 32 - 35 - 15 - 5 - 10 - 4.908425 -
4 - 000010011010010101 35 - 7 - 6 - 5 - 10 - 4.970696 -
5 - 001011111010100001 35 - 7 - 6 - 5 - 10 - 4.673993 -
6 - 010011011010101100 24 - 27 - 18 - 5 - 10 - 4.967033 -
7 - 001001000011100010 24 - 27 - 18 - 5 - 10 - 4.725275 -
8 - 001011101011110110 20 - 39 - 12 - 5 - 10 - 4.835165 -
```

```
9 - 000011101010010100 20 - 39 - 12 - 5 - 10 - 4.963370 -
(...)
30 - 001001000010100010 26 - 21 - 5 - 5 - 10 - 4.000000 -
31 - 001011000010110100 26 - 21 - 5 - 5 - 10 - 4.908425 -
32 - 001011001010110100 2 - 16 - 8 - 5 - 10 - 4.410256 -
33 - 000011101011100100 2 - 16 - 8 - 5 - 10 - 4.501832 -
34 - 010010011010110100 10 - 38 - 18 - 5 - 10 - 4.611722 -
35 - 001001001010110111 10 - 38 - 18 - 5 - 10 - 4.904762 -
36 - 001001000011100000 27 - 22 - 15 - 5 - 10 - 4.981685 -
37 - 011001000010100010 27 - 22 - 15 - 4 - 5 - 3.486486 -
38 - 000010010010110100 34 - 10 - 1 - 5 - 10 - 4.791209 -
39 - 011011111010010101 34 - 10 - 1 - 4 - 5 - 3.000000 -
-------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 4.420651
Soma Ajuste: 176.826050
Numero Recombinacoes.: 933
Numero Mutacoes: 941
Enter para continuar

GERACAO NUMERO: 71
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: -88
Divisor: 181
Dados Invalidos: numeros negativos.

RELATORIO DE POPULACAO: Geracao Numero: 130

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
-------------------------------------------------------------------------------------------
0 - 010101101010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 001001101010100001 7 - 21 - 2 - 5 - 10 - 5.000000 -
2 - 011011000010110000 38 - 5 - 13 - 4 - 5 - 3.583333 -
3 - 010001000001100100 38 - 5 - 13 - 4 - 5 - 3.625000 -
4 - 011010101010111000 22 - 3 - 13 - 4 - 5 - 3.697917 -
5 - 011010100010100101 22 - 3 - 13 - 4 - 5 - 3.510417 -
6 - 010101111011110100 13 - 8 - 18 - 5 - 10 - 4.800000 -
7 - 011001100011110100 13 - 8 - 18 - 5 - 10 - 4.563636 -
8 - 011011000010110100 9 - 37 - 1 - 4 - 5 - 3.625000 -
9 - 000100101010100100 9 - 37 - 1 - 5 - 10 - 4.993939 -
(...)
31 - 001011010010110100 38 - 12 - 6 - 5 - 10 - 4.830303 -
32 - 011010100010100000 3 - 19 - 5 - 4 - 5 - 3.458333 -
33 - 011000100010101000 3 - 19 - 5 - 4 - 5 - 3.708333 -
34 - 011010100010110100 3 - 37 - 11 - 4 - 5 - 3.666667 -
35 - 001011000011101000 3 - 37 - 11 - 5 - 10 - 4.800000 -
36 - 010001000000101000 5 - 9 - 18 - 4 - 5 - 3.000000 -
37 - 010000101010100100 5 - 9 - 18 - 5 - 10 - 4.757576 -
38 - 011001100010000110 7 - 29 - 18 - 4 - 5 - 3.270833 -
39 - 101011000010110101 7 - 29 - 18 - 2 - 11 - 1.000000 -
```

```
--------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 4.033016
Soma Ajuste: 161.320633
Numero Recombinacoes.: 1737
Numero Mutacoes: 1739
Enter para continuar

GERACAO NUMERO: 131
Avaliando Individuo Numero: 0
Dividendo: Divisor:
Dividendo: 173
Divisor: 180
Quociente: 0
Resto: 173
(...)
Avaliando Individuo Numero: 39
Dividendo: Divisor:
Dividendo: 47
Divisor: 182
Quociente: 0
Resto: 47

RELATORIO DE POPULACAO: Geracao Numero: 146

Individuos da populacao
N. Indiv. - Cromossomo - Pai1 - Pai2 - P. Cross. -  N. nos corretos - no desvio - ajuste
--------------------------------------------------------------------------------------------
0 - 0101011010110100 0 - 0 - 0 - 5 - 10 - 5.000000 -
1 - 0001011110001100 36 - 18 - 13 - 5 - 10 - 5.000000 -
2 - 0100011110110110 25 - 23 - 13 - 5 - 10 - 4.533333 -
3 - 0011010110110111 25 - 23 - 13 - 5 - 10 - 4.900000 -
4 - 0001011110001100 1 - 2 - 18 - 5 - 10 - 4.508333 -
5 - 0010011100111110 1 - 2 - 18 - 5 - 10 - 4.504167 -
6 - 0101011110110101 35 - 25 - 3 - 5 - 10 - 4.954167 -
7 - 0100010010101100 35 - 25 - 3 - 5 - 10 - 4.679167 -
8 - 0010111100110100 38 - 39 - 9 - 5 - 10 - 4.250000 -
9 - 0011011110001000 38 - 39 - 9 - 4 - 5 - 3.663102 -
(...)
30 - 0100011110110111 25 - 32 - 16 - 5 - 10 - 4.887500 -
31 - 0110110110100101 25 - 32 - 16 - 4 - 5 - 3.732620 -
32 - 0001011110111000 31 - 14 - 18 - 5 - 10 - 4.754167 -
33 - 0001011110001100 31 - 14 - 18 - 4 - 5 - 3.983957 -
34 - 0010011011101100 33 - 30 - 15 - 2 - 11 - 1.054945 -
35 - 0100011100110100 33 - 30 - 15 - 5 - 10 - 4.000000 -
36 - 0010011000000000 2 - 15 - 2 - 4 - 5 - 3.764706 -
37 - 0010011100110110 2 - 15 - 2 - 5 - 10 - 4.904167 -
38 - 0001011110001100 14 - 23 - 18 - 4 - 5 - 3.983957 -
39 - 0001011110110110 14 - 23 - 18 - 5 - 10 - 4.412500 -
--------------------------------------------------------------------------------------------
Estatisticas da Geracao e Valores Acumulados

Max. Ajuste: 5.000000
Min. Ajuste: 1.000000
Media Ajuste: 4.137643
Soma Ajuste: 165.505737
Numero Recombinacoes.: 1936
Numero Mutacoes: 1935
Enter para continuar
```

```
DataGen - Identificada situacao de possivel nao executabilidade do caminho pretendido.

Numero da Geracao: 146
Numero da Tentativa: 5840

O ajuste medio da populacao teve progresso acumulado inferior a 0.30
e progresso sucessivo inferior a 0.50 nas ultimas 102.000000 geracoes.

Progresso acumulado em 102.000000 geracoes: 0.061229

Caminho Pretendido:  1  2  3  4  6  7  8  9  6  7  9  6  7  8  9  6  7  9  6  10  12  13

Numero de nos corretamente executados da melhor solucao: 5
Ultimo no corretamente executado: 6
Verifique executabilidade do arco ( 6 , 7 ) em relacao aos predicados anteriores do caminho.

Interrompe a busca ? (s|n) s
DataGen - Busca interrompida: Nao executabilidade identificada.
```