LUCIANO PETINATI FERREIRA

TDSGEN - UMA FERRAMENTA DE GERAÇÃO DE DADOS DE TESTE BASEADA EM ALGORITMOS GENÉTICOS

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

Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio

CURITIBA

2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Luciano Petinati Ferreira, avaliamos o trabalho intitulado, *"TDSGEN - UMA FERRAMENTA DE GERAÇÃO DE DADOS DE TESTE BASEADA EM ALGORITMOS GENÉTICOS"*, cuja defesa foi realizada no dia 28 de novembro de 2003, às quatorze horas, no Anfiteatro A do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 28 de novembro de 2003.

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

Prof. Dr. Edmundo Sérgio Spoto
Fundanet/Marília - Membro Externo

Prof. Dr. Martin A. Musicante
DINF/UFPR - Membro Interno

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

AGRADECIMENTOS

Aos meus pais, José e Rosa, e irmãos, Rogerio e André, que deram todo o apoio e torcida neste desafio, a quem dedico este trabalho.

Aos meus amigos que por mais distantes que estiveram, torciam por mim, entendiam minha ausência em alguns momentos e até ajudaram a ficar acordado estudando. Em especial meus amigos Dariano, Ibra, Guile, Leo e Mari.

Aos companheiros de profissão que sempre que puderam, me ouviram e deram dicas para o desenvolvimento do trabalho. Em especial agradeço ao Auri e Juliano que me ajudaram bastante, os amigos Dino e Delamaro que até o fim torciam para a conclusão do trabalho e aos irmãos Ari e Horácio que muito me apoiaram.

Aos profissionais e amigos do departamento de informática da UFPR.

Finalmente, agradeço à minha orientadora Prof^a. Dr^a. Silvia Regina Vergilio, a principal responsável pela elaboração e conclusão deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A atividade de teste é fundamental no ciclo de vida de desenvolvimento de software para encontrar defeitos e garantir o aumento da qualidade do software. Para auxiliar a etapa de seleção de dados de teste, diferentes critérios de teste foram propostos e exigem que certas condições ou elementos sejam exercitados durante o teste. Algumas ferramentas foram então desenvolvidas para a aplicação de tais critérios, entre elas a POKE-TOOL e PROTEUM, que apoiam os critérios estruturais e baseados em erros, respectivamente. Essas ferramentas dão suporte à utilização dos critérios, mas a completa automação da atividade de teste é impossível.

A geração de dados para satisfazer um determinado critério é uma tarefa difícil, e possui inúmeras limitações inerentes à própria atividade de teste. Para minimizar essas limitações, alguns autores propuseram o uso de algoritmos meta-heurísticos tais como Algoritmos Genéticos (AG), originando um novo campo de pesquisa chamado Teste Evolucionário [21].

A idéia básica de um AG é evoluir soluções para um dado problema utilizando-se de conceitos da Teoria da Evolução de Darwin. Mas a maioria desses trabalhos não oferece um ambiente de teste integrando AG's com diferentes ferramentas de teste.

Este trabalho descreve a Ferramenta *TDSGen* que integra as Ferramentas POKE-TOOL e PROTEUM, e utiliza-se de AG para a geração de dados de teste para satisfazer critérios estruturais e baseados em erros, proporcionando um ambiente que suporta a completa aplicação de estratégias de teste, diferenciando-a de outras ferramentas existentes. A *TDSGen* também permite a utilização de AG com diferentes estratégias e hibridização com o objetivo de aumentar o desempenho.

Um experimento é apresentado para validar a abordagem implementada para a geração de dados de teste e os resultados mostram que as estratégias implementadas na ferramenta são muito promissoras e contribuem para um aumento da cobertura dos critérios com relação à geração aleatória e a geração baseada em AG simples.

ABSTRACT

The testing activity is a fundamental phase in the software enginering process to ensure software quality and to reveal faults in the program. To aim at the selection of test data, several testing criteria were proposed. They generally require the exercising of certain conditions or elements in the program being tested. Some tools were developed to support such criteria; POKE-TOOL and PROTEUM are examples of testing tools that implement respectivelly structural and fault-based testing criteria. These tools, however, do not support the complete automation of the testing activity.

The automation of the test data generation to satisfy a given criterion is not possible, due to several testing limitations. To overcome these limitations, some authors proposed the use of meta-heuristic algorithms, such as Genetic Algorithms (GA), originating a new search field called Evolutionary Testing [21].

The GA has the goal of evolving solutions for a given problem by appling the concepts of Darwin's Evolutionary Theory. However, most of these works do not offer a testing environment and are not integrated to testing tools.

This work describes a tool, called *TDSGen*, that integrates the testing tools mentioned above: POKE-TOOL and PROTEUM. *TDSGen* is based on GA and generates test data to satisfy structural and fault based criteria. It is a support tool for a testing strategy that includes the application of different criteria, differently of the other existent tools. *TDSGen* tool also allows the use of hybrid GA with different strategies to improve performance.

An experiment is presented to validate the test data generation approach implemented and the results are very promissing. The implemented approach improved the obtained coverage for the criteria when compared to randon and simple GA based generations.

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

O teste tem se tornado uma atividade importante no ciclo de vida de desenvolvimento de software. Sua aplicação é fundamental para encontrar defeitos e garantir o aumento da qualidade do software. Uma forma de testar a corretitude de um programa, é a execução do mesmo sobre todos os dados do conjunto de entrada e verificar a resposta produzida, porém isso é custoso [16, 44] e se torna inviável.

A solução para este problema é o planejamento do teste de forma a selecionar dados que aumentem a produtividade da atividade de teste. Para auxiliar esse planejamento, diferentes técnicas de teste podem ser aplicadas. As principais técnicas de teste existentes consideram diversos aspectos para auxiliar a geração de dados de teste, e podem revelar diferentes tipos de defeitos [44]. Associados a estas técnicas existem diferentes critérios de teste. Os critérios de teste são predicados a serem satisfeitos para se considerar a atividade de teste encerrada. Eles guiam a atividade de teste fornecendo diretrizes para a seleção de dados de teste e fornecem medidas de cobertura para avaliar um conjunto de casos de teste. Eles exigem que certos elementos do programa sejam exercitados e quando todos os elementos requeridos forem cobertos, diz-se que o critério foi satisfeito. Abaixo estão relacionados os principais critérios, agrupados segundo as características que eles consideram para derivar os dados de teste.

- Critérios Funcionais - consideram aspectos da funcionalidade do programa para derivar os dados de teste. Exemplos: Particionamento em Classes de Equivalência, Análise de Valor Limite e Técnicas de Grafo de Causa-Efeito [44];

- Critérios Estruturais - consideram aspectos internos da estrutura do software ou da implementação. São divididos em:

  - Critérios baseados em fluxo de controle [22]: Todos-Nós, Todos-Arcos e Todos-Caminhos;
  - Critérios baseados em fluxo de dados [22, 45]: Todos-Potenciais-Usos, Todos-Potenciais-Du-Caminhos, Todos-Potenciais-Usos/Du, etc;

- Critérios Baseados em Erros - consideram os principais erros comuns ao desenvolvimento do software. Exemplo: Análise de Mutantes [41] e Semeadura de Erros [49].

Buscar dados de um conjunto de entrada que exercitem os elementos requeridos, exige do testador conhecimentos específicos do critério utilizado além de poder exigir uma minuciosa análise do código do programa em teste, o que pode ser agravado à medida que o programa possui tamanho e complexidade aumentados, podendo encarecer a aplicação dos critérios de teste.

Para auxiliar a aplicação desses critérios, esforços têm sido aplicados no desenvolvimento de ferramentas para a automação da geração de dados de teste que satisfaçam os elementos exigidos pelos critérios de teste [12, 26, 35]. Exemplos dessas ferramentas são: ASSET [12], que implementa os critérios de Rapps e Weyuker [45] para a Linguagem Pascal, ATAC [17] apoia os mesmos critérios que ASSET, porém para a Linguagem *C*, POKE-TOOL que implementa os critérios Todos-Potenciais-Usos [35], MOTHRA [40] e PROTEUM [34] que apoiam a Análise de Mutantes para as Linguagem FROTRAN e *C* respectivamente.

Essas ferramentas dão suporte à utilização dos critérios, mas a completa automação da atividade de teste é impossível. A dificuldade da automação da atividade de teste, mais especificamente geração de dados de teste, é devida a problemas de indecidibilidade como [22]:

- não existe algoritmo de propósito geral para determinar um conjunto de casos de teste T que satisfaça um determinado critério de teste C, nem mesmo para saber se T existe [45].

- não existe procedimento para comprovar a corretude de um programa;

- é indecidível determinar se dois programas computam a mesma saída;
- é indecidível saber se um determinado caminho é ou não executável e;
- é indecidível detectar ocorrência de correção coincidente, onde um item incorreto coincidentemente produz um resultado correto.

Técnicas para geração de dados de teste podem ser classificadas em: 1) geração aleatória [27]; 2) geração com base em execução simbólica [8, 30, 51]; 3) geração baseada em execução dinâmica [3]. Essas técnicas têm sido aplicadas com diferentes graus de sucesso, devido às limitações da atividade de teste mencionadas acima. Para lidar com essa situação, alguns autores propuseram o uso de algoritmos meta-heurísticos tais como Algoritmos Genéticos (AG), Busca Tabu, entre outros, originando um novo campo de pesquisa chamado Teste Evolucionário [21].

A idéia básica de um AG [25] é evoluir soluções para um dado problema utilizando-se de conceitos da Teoria da Evolução de Darwin, tais como, crossover, mutação, etc. Na literatura podem ser encontrados muitos trabalhos que exploram o uso de AG para geração de dados de teste. Esses trabalhos, porém, focalizam critérios específicos como: os trabalhos que abordam critérios estruturais baseados em fluxo de controle [5, 13, 15, 33, 39, 42], o trabalho de Wegener para critérios estruturais baseados em fluxo de dados [21] e os trabalhos para critérios baseados em erros [19, 29]. A maioria desses trabalhos não oferece um ambiente para apoiar a organização e a completa aplicação de um ou mais critério de teste e também não esta integrada a uma ferramenta [21].

Segundo Wegener [21], os trabalhos sobre teste evolucionário podem ser divididos em duas categorias, dependendo da forma como a função de *fitness* [1] é calculada. Na primeira, a função de *fitness* de cada indivíduo é calculada analisando uma medida de cobertura do caso de teste correspondente [2, 33, 37, 39, 42, 46]. Na segunda, a função de *fitness* é orientada por um objetivo [7, 13, 15] que geralmente é a cobertura de um dado elemento requerido. Os trabalhos nessa categoria surgiram com o objetivo de melhorar o desempenho dos AG's da primeira categoria. No entanto, eles requerem a análise do

[1] Função de avaliação de Algoritmos Genéticos e esta descrita no Capítulo 2.

programa em teste e são mais custosos.

Para melhorar o desempenho de um AG, principalmente para problemas complexos, alguns estudos foram conduzidos e estratégias foram propostas bem como o uso de AG Híbridos [31]. Esses estudos exploram heurísticas tais como: Busca Tabu e técnicas de criação de nichos como Crowding [28] e Sharing [9]. Mas essas estratégias têm sido pouco exploradas na geração de dados de teste.

1.2 Motivação

Os pontos principais para o desenvolvimento deste trabalho são:

- A necessidade e a importância que a atividade de teste assume no processo de desenvolvimento de software para garantia de qualidade;
- As dificuldades e custos da aplicação da atividade de teste;
- A influência dos casos de teste para a atividade de teste;
- A necessidade de pesquisa para desenvolvimento de técnicas e ferramentas que melhorem a geração automática de dados de teste;
- A inexistência de um ambiente de teste baseado em AG que apoie a aplicação completa de uma estratégia de teste incluindo critérios diferentes e que seja integrado às outras ferramentas de teste;
- Desenvolvimento de pesquisa sobre combinações de técnicas de hibridização em AG na busca por melhores resultados. Possibilidade de explorar essas técnicas na atividade de geração de dados de teste.

1.3 Objetivos

Devido ao exposto acima, este trabalho utiliza a abordagem de Algoritmos Genéticos (AG), métodos de busca e otimização baseados nos processos naturais de evoluação [5, 14, 33], com algumas modificações incrementais(hibridização) e implementação de algumas

estratégias para a geração de um conjunto de dados de teste para satisfação dos seguintes critérios de teste: 1) critérios estruturais, implementados pela ferramenta de teste POKE-TOOL [35]: critérios baseados em fluxo de controle (Todos-Nós e Todos-Arcos) e critérios baseados em fluxo de dados (Todos-Potenciais-Usos, Todos-Potenciais-Du-Caminhos e Todos-Potenciais-Usos/Du); e 2) critério baseado em erros, implementado pela ferramenta Proteum [34]: Análise de Mutantes.

A principal contribuição desse trabalho é a implementação de uma ferramenta, chamada *TDSGen* (**T**est **D**ata **S**et **Gen**erator), diferenciada das demais existentes por possibilitar um ambiente que permite a aplicação completa de estratégias de teste para critérios estruturais e baseados em erros. Também são apresentados dados de experimentos sobre a abordagem de geração de dados de teste proposta, possibilitando assim, estudar a influência de outras estratégias e de hibridização em AG no teste evolucionário.

1.4 Organização

O Capítulo 2 introduz Algoritmos Genéticos, algoritmo básico, operadores e estratégias. O Capítulo 3 aborda o teste de software, terminologias, principais técnicas, critérios e ferramentas. O Capítulo 4 trata da geração de dados de teste, seus paradigmas e trabalhos da literatura sobre a geração de dados usando Algoritmos Genéticos. O Capítulo 5 descreve a ferramenta *TDSGen*, que implementa AG híbrido, seus principais aspectos de implementação e funcionamento. O Capítulo 6 apresenta os resultados dos experimentos de avaliação da ferramenta. O Capítulo 7 finaliza o trabalho com as conclusões e trabalhos futuros. O trabalho contém um apêndice, Apêndice A, que apresenta o fluxo de execução da *TDSGen*.

CAPÍTULO 2

ALGORITMOS GENÉTICOS

Os Algoritmos Genéticos (AG) formam a parte da área dos Sistemas Inspirados na Natureza. Foram criados por John Holland [25], que tinha como meta inicial, estudar formalmente o fenômeno da adaptação, como ocorre na natureza e desenvolver uma forma de importá-lo para sistemas computacionais [32]. Algoritmos Genéticos são métodos generalizados de busca e otimização que simulam os processos naturais de evolução, aplicando a idéia da seleção natural de Charlles Darwin. Basicamente, a idéia de Darwin diz que indivíduos melhor adaptados de uma população, no momento da reprodução passarão suas características, através de seu material genético, podendo assim formar indivíduos melhor adaptados.

Para a aplicação de Algoritmos Genéticos é necessário:

1. Representar o espaço de busca do problema (possíveis soluções);
2. Uma função de avaliação (função de *fitness*) para determinar o quanto uma resposta é boa para o problema;
3. Representar as soluções na forma computacional.

A vantagem de se utilizar Algoritmos Genéticos é o fato de proporcionar a utilização de um rico espaço de busca simultaneamente, podendo encontrar muitos pontos bons para a solução evitando ótimos locais.

O funcionamento (Diagrama de blocos) de um algoritmo genético é apresentado na Figura 2.1.

O algoritmo genético começa com a geração da população inicial. Inicia-se então, um ciclo de evolução até que uma solução seja encontrada ou um número máximo de iterações seja alcançado. Esse ciclo de evolução é composto por:

- avaliar cada indivíduo da população;

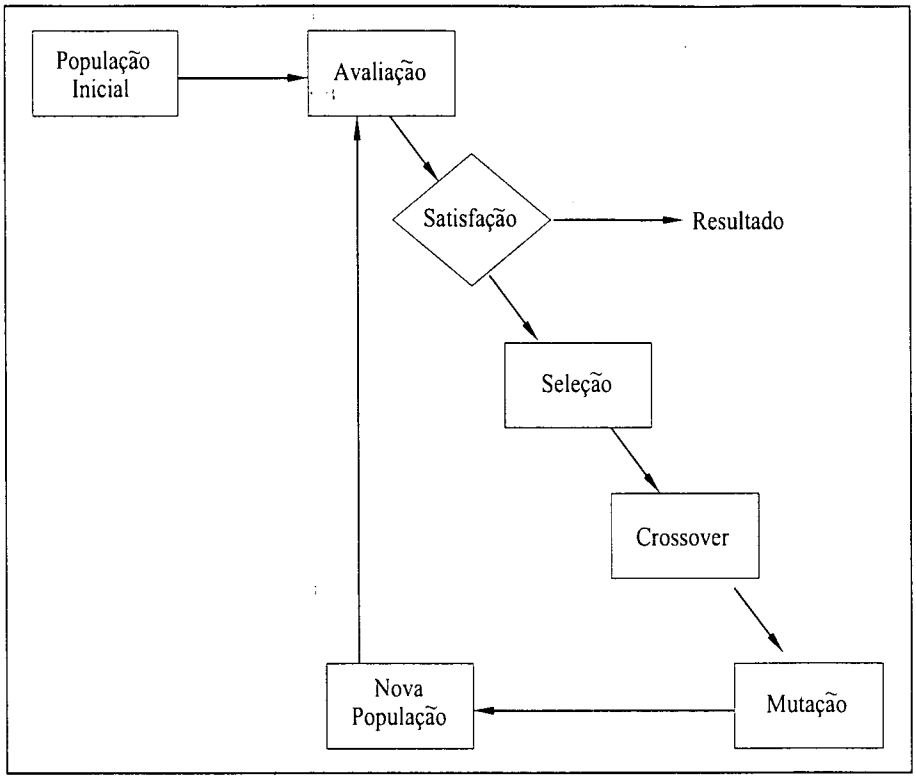

Figura 2.1: Diagrama de blocos de um Algoritmo Genético

- selecionar os indivíduos, com base na avaliação, para criar a nova população;
- aplicar os operadores genéticos.

Conforme [32], não existe uma definição rigorosa de Algoritmos Genéticos, aceita por todos da comunidade de computação evolucionária, que os diferencie de outros métodos de computação evolucionária. Entretanto, pode-se dizer que a maioria dos métodos chamados de Algoritmos Genéticos possuem pelo menos os seguintes elementos em comum: população de cromossomos, seleção com base na função de *fitness*, operador de crossover e operador de mutação.

Nas seções a seguir além desses elementos em comum, são detalhados os pontos principais dos algoritmos genéticos e uma definição para Algoritmos Genéticos Híbridos é apresentada.

2.1 Indivíduo

Para a aplicação de Algoritmos Genéticos a um problema particular, é necessário projetar o mapeamento de potenciais soluções para o problema, o que geralmente ocorre na forma de uma seqüência de bits ou caracteres, que representem os atributos da solução. Essa seqüência de bits forma um indivíduo. A tarefa de mapeamento nem sempre é fácil e pode envolver heurísticas adicionais como codificadores [52].

Fazendo uma analogia à genética, o indivíduo é um cromossomo do DNA onde cada posição na seqüência de bits representa a presença ou a ausência de uma determinada característica.

2.2 População de Cromossomos

A população é o conjunto de indivíduos que estão sendo cogitados como solução do problema, ou seja, é o domínio das possíveis soluções para o problema sendo tratado. Um método comumente utilizado para gerar a população incial do AG é a geração aleatória dos indivíduos.

O tamanho da população é constante durante a execução do algoritmo genético e cada execução constitui uma geração.

2.3 *Fitness*

A função de *fitness* ou medida de adaptação de um indivíduo, é tipicamente definida como a probabilidade que o mesmo tem para sobreviver e reproduzir. O indivíduo que possui um bom escore de *fitness* pode ser interpretado como um indivíduo que resolve bem o problema em questão [32].

2.4 Seleção

O processo de seleção é um ponto importante a ser decidido na aplicação de Algoritmos Genéticos pois selecionará os indivíduos para formarem a próxima população.

Tabela 2.1: Dados exemplos para método de seleção por roleta

Indivíduo	Representação	*fitness*	Percentual por *fitness*
A	10010000100	3	15%
B	11110111111	10	50%
C	11000000110	4	20%
D	00000001100	2	10%
E	00000001000	1	5%

A proposta da seleção deve levar em conta o fator de adaptação (*fitness*) dos indivíduos para que os mais adaptados tenham maior chance de serem selecionados para formar descendentes melhor adaptados.

A seleção pode ser responsável por privilegiar os indivíduos de alto escore de *fitness*, fazendo com que estes dominem a população reduzindo a diversidade da espécie e proporcionando uma convergência mais rápida para a solução, como também pode fazer o contrário.

Em geral, os Algoritmos Genéticos utilizam um método de seleção para obter um par de indivíduos que se tornarão os pais de dois novos indivíduos para a nova geração.

Entre os métodos de seleção mais usados estão: Roleta, Ranking e Torneio [32].

2.4.1 Método de Seleção da Roleta

Cada indivíduo da população recebe uma fatia de uma roleta circular proporcional ao seu escore obtido com a função de *fitness*. Assim, ao rodar a roleta, os indivíduos com maior escore de *fitness* ganham maior probabilidade de serem escolhidos.

Como já mencionado anteriormente, o tamanho da população é constante e a cada par de indivíduos selecionados, o algoritmo gera dois novos indivíduos para fazer parte da população da próxima geração. Com isso, o método da roleta deve ser utilizado N vezes para manter o tamanho da população inalterado, onde N é tamanho da população. Cada vez que este método de seleção é utilizado, um indivíduo é escolhido na população.

A Figura 2.2 ilustra o funcionamento deste método de seleção. Para o exemplo, o escore de *fitness* foi calculado com base na quantidade de número 1 na representação binária do indivíduo e os resultados são apresentados na Coluna *fitness* da Tabela 2.1.

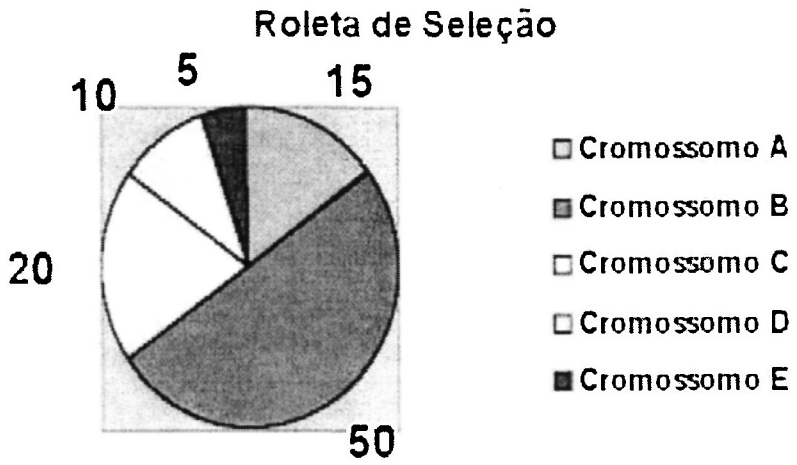

Figura 2.2: Ilustração do método de seleção por roleta

2.4.2 Método de Seleção por Ranking

Esse método de seleção tem o propósito de evitar uma rápida convergência para uma solução. Na versão proposta por Baker [24], os indivíduos da população são ordenados de acordo com seu escore de *fitness*, e a chance que o indivíduo tem de ser selecionado é dada pela posição no ranking, não mais pelo valor absoluto do escore de *fitness*. Esse método de seleção elimina a grande vantagem proporcionada aos indivíduos que possuem um escore muito alto, previnindo assim a convergência acelerada, o que é equivalente a diminuir a pressão de seleção. Um aspecto negativo desse método é que em alguns casos, pode não permitir distinguir se um indivíduo é mais saudável que o seu competidor mais próximo.

A Tabela 2.2 ilustra a aplicação do método de seleção por Ranking e realiza uma comparação com o método de seleção por Roleta. A tabela mostra que o melhor induvíduo (B) teve sua probabilidade reduzida de 50% para 32.5%.

2.4.3 Método de Seleção por Torneio

A seleção por torneio também elimina o efeito da pressão de seleção proporcionado pelo método da roleta. Conforme [32], esse método seleciona dois indivíduos da população

Tabela 2.2: Exemplo para método de seleção por Ranking e comparação com Roleta

Indivíduo	Representação	*fitness*	Percentual por *fitness*	Rank	Percentual por Rank
A	10010000100	3	15%	3	19.5%
B	11110111111	10	50%	5	32.5%
C	11000000110	4	20%	4	26%
D	00000001100	2	10%	2	13%
E	00000001000	1	5%	1	6.5%

de forma aleatória. Depois é escolhido, também aleatoriamente, um número R entre zero e um. Se R for maior que um parâmetro K, o indivíduo de melhor escore de *fitness* é selecionado para ser pai, caso contrário o outro indivíduo será selecionado. O valor de K é um parâmetro que deve ser informado para o algoritmo de seleção.

2.5 Operadores Genéticos

Outra decisão na aplicação de Algoritmos Genéticos é sobre a escolha de quais operadores genéticos serão utilizados. Os operadores genéticos servem para aumentar a variedade para a próxima geração. Os operadores mais comuns são Crossover e Mutação e estão apresentados a seguir.

2.5.1 Crossover

O operador de crossover em sua versão mais simples, conhecido como crossover de ponto único, seleciona aleatoriamente uma posição do cromossomo e troca as informações de dois cromossomos, previamente selecionados, a partir deste ponto criando dois novos indivíduos. A Figura 2.3 ilustra a aplicação do operador sobre dois indivíduos de representação binária.

Uma outra versão deste operador genético é o crossover de dois pontos onde duas posições do cromossomo são escolhidas aleatoriamente e o segmento entre esses pontos é trocado. Este operador é ilustrado na Figura 2.4.

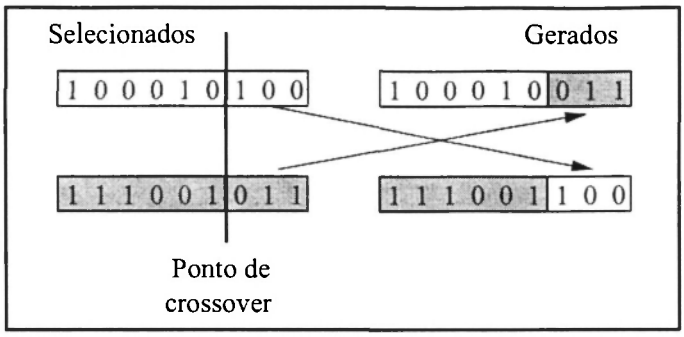

Figura 2.3: Ilustração do crossover de ponto único

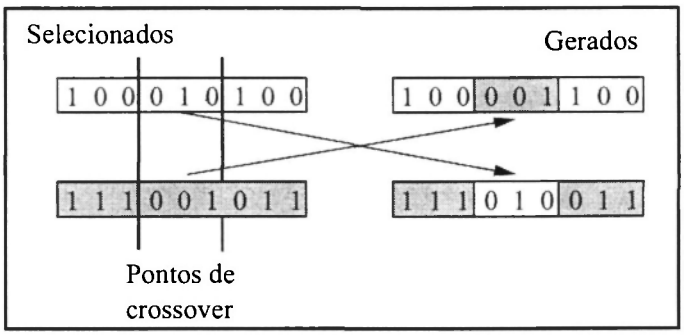

Figura 2.4: Ilustração do crossover de dois pontos

2.5.2 Mutação

Este operador escolhe aleatoriamente um dos bits de um cromossomo/indivíduo, e troca o seu valor. A Figura 2.5 ilustra a aplicação do operador em um indivíduo de representação binária.

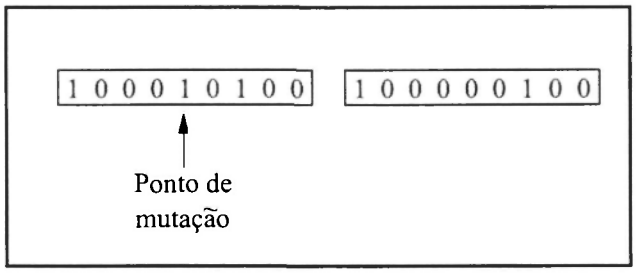

Figura 2.5: Ilustração de mutação

Na aplicação de Algoritmos Genéticos são utilizados parâmetros genéticos para decidir

sobre suas funcionalidades e características. Alguns desses parâmetros são: tamanho da população, número máximo de gerações, taxa de crossover e taxa de mutação.

2.6 Estratégias Aplicadas à AG's

Algoritmos Genéticos podem usar técnicas no processo de evolução para modificar seu comportamente e resultado. Esta seção apresenta algumas estratégias que podem ser aplicadas em AG's para se obter um melhor desempenho.

2.6.1 Elitismo

Esta técnica foi proposta por De Jong [28] e garante a presença dos melhores indivíduos da população atual na população da próxima geração. Como AG's são estocásticos, existe a probabilidade de que bons indivíduos não sejam selecionados para reprodução. Para evitar esta perda, o indivíduo com melhor escore de *fitness* é copiado para a próxima população.

2.6.2 Eliminação de Duplicatas

Dependendo do domínio da aplicação, a duplicidade de indivíduos (que ocasiona a perda de diversidade da população) pode ser prejudicial para o resultado final. Para resolver este problema, algumas implementações de AG's ignoram os indivíduos duplicados, que foram criados no processo de evolução, gerando um novo indivíduo para subistituí-lo [31].

2.6.3 Hibridização

Além dos AG's tradicionais, outros tipos foram desenvolvidos combinando algumas técnicas heurísticas para melhorar o desempenho dos AG's na solução de problemas mais específicos e complexos. Este processo é a hibridização e os resultados obtidos mostram-se melhores que o uso das técnicas utilizadas isoladamente [31].

As próximas seções apresentam as técnicas de hibridização mais utilizadas: Busca Tabu e compartilhamento de recursos - *sharing*.

2.6.3.1 Busca Tabu

É uma meta-heurística para busca local que se utiliza de estruturas especializadas de memória para evitar problemas com ótimos locais e alcançar um resultado balanceado entre intensidade e diversidade na busca [6].

Da forma como se encontra hoje, foi apresentada inicialmente por Glover [10]. Esta técnica consiste em obter uma solução j a partir de uma solução atual i, através de uma operação/movimento que transforma i em j. Esta solução j pode ser vista como uma solução vizinha de i.

Após obtida a solução j, uma condição de término/satisfação é avaliada para decidir se o processo continua a busca por outra solução j ou pode parar.

A busca Tabu implementa uma lista, chamada Lista Tabu, que proporciona um mecanismo de memorização, evitando que o processo de busca entre em um ciclo repetitivo [1].

A avaliação de um indivíduo caracteriza um tabu [1] evitando assim, que a avaliação se repita a menos que satisfaça condições para ignorar esta restrição, chamadas de restrições Tabu. O conjunto de condições analisadas para ignorar uma restrição Tabu chama-se critério de aspiração [10].

Como descrito, alguns elementos podem ser identificados para Busca Tabu, são eles:

- Movimentos: operadores que transformam uma solução em outra - solução vizinha.
- Lista Tabu: um mecanismo usado para manter todas as soluções anteriores avaliadas. É também conhecida como lista de restrições.
- Critério de Aspiração: função para determinar quando uma restrição Tabu deve ser ignorada, decidindo que uma solução (movimento) deve ser processada.
- Término: função para definir quando o processo de Busca Tabu deve ser finalizado que geralmente ocorre quando não existe mais movimentos possíveis a serem realizados ou um número máximo de iterações é alcançado.
- Parâmetros de configuração: parâmetros são usados para ajustar a Busca Tabu. Entre eles estão o número máximo de iterações, o tamanho da Lista Tabu, regras

de parada e nível de aspiração.

Uma representação de como é o funcionamento da Busca Tabu é apresentada na Figura 2.6 [4].

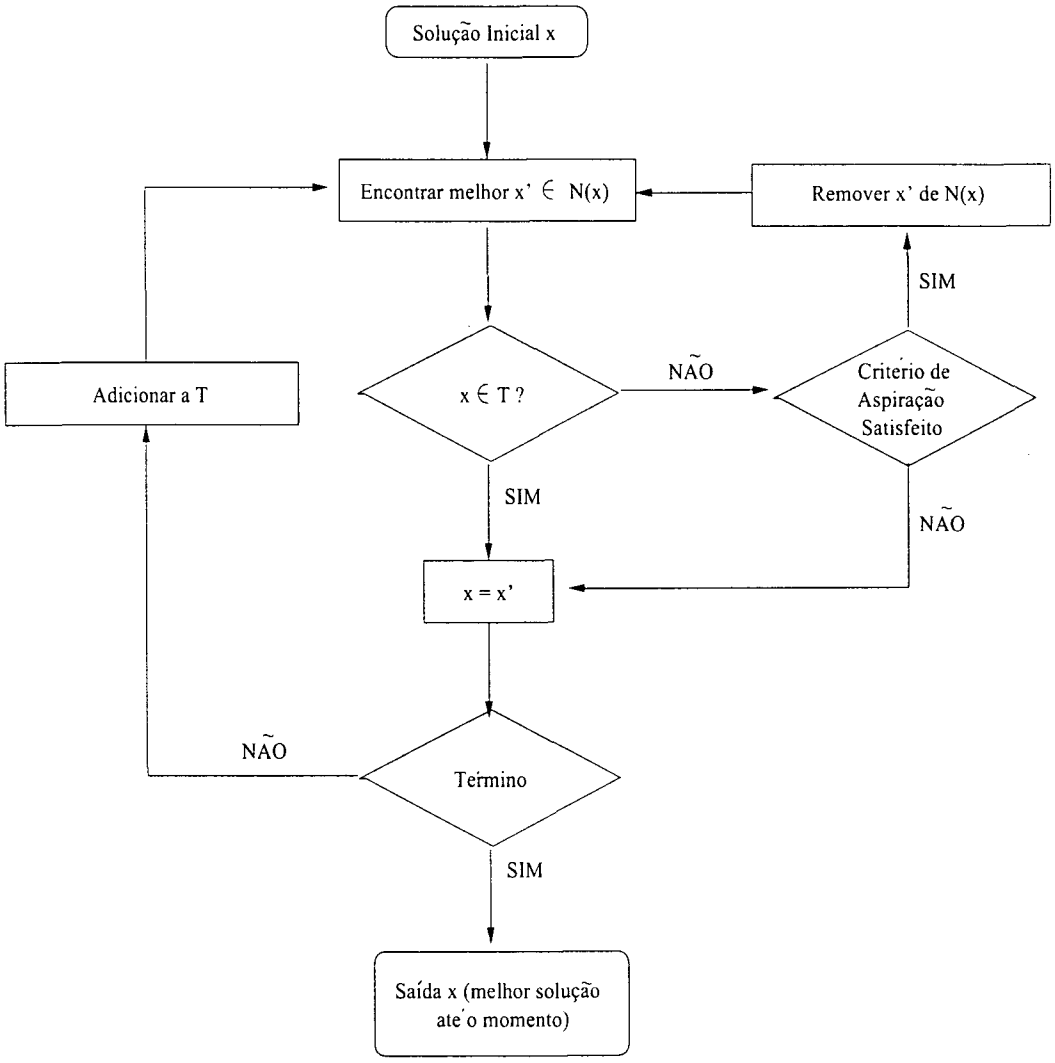

Figura 2.6: Método de Busca Tabu

2.6.3.2 *Sharing*

O método de compartilhamento de recursos, conhecido como *sharing*, foi introduzido por Goldberg e Richardson. O objetivo deste método é reduzir o valor de aptidão de indivíduos similares dentro de uma população [9].

Esta técnica é aplicada para prejudicar a chance de selecionar indivíduos idênticos ou similares no processo evolutivo do AG, forçando o aumento da diversidade populacional [9].

A técnica de *sharing* consiste em obter uma medida de similaridade entre os indivíduos da população e penalizar o valor de aptidão dos indivíduos conforme essa medida de similaridade. Quanto maior a medida de similaridade de um determinado indivíduo com relação aos outros indivíduos da população, maior a redução em seu valor de aptidão - função de *fitness*.

Uma forma prática para obter a medida de similaridade é a função *Sharing* Triangular, ilustrada na Figura 2.7.

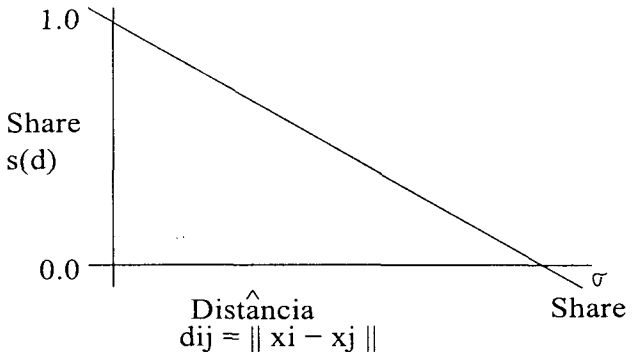

Figura 2.7: Função *Sharing* Triangular

A medida de similaridade de um indivíduo é calculada somando-se o resultado da função de *sharing* para todos os outros indivíduos da população. [9] Este valor pode variar de 0.0, indicando um indivíduo sem outro similar na população, até 1.0, indicando que um ou mais indivíduos da população são idênticos a este.

Calculada a medida de similaridade, basta intervir no processo normal de avaliação dos indivíduos em um AG tradicional, punindo o valor de aptidão dos indivíduos com sua respectiva medida de similaridade, conforme ilustra a Fórmula 2.1.

$$F_s(x_i) = \frac{F(x_i)}{\sum_{j=1}^{I} s(d(x_i, x_j))} \tag{2.1}$$

Onde $F(x_i)$ é o valor da função de fitness de um indivíduo, $s(d(x_i, x_j))$ é a soma limiar

de distâncias entre dois indivíduos, I o tamanho da população menos um [9].

O resultado final é que o crescimento descontrolado de espécies dentro de uma população será dificultado proporcionando a diversidade da população.

2.7 Considerações Finais

Neste capítulo foram apresentados os principais conceitos sobre AG's, seu algoritmo básico e funcionamento.

As estratégias descritas na Seção 2.6 são bastante importantes pois podem aumentar o desempenho de um AG. Com esse objetivo, algumas dessas estratégias foram implementadas na Ferramenta *TDSGen*, descrita no Capítulo 5.

CAPÍTULO 3

TESTE DE SOFTWARE

Pressman [44] diz que a atividade de teste de software é um elemento crítico para garantia de qualidade de software; segundo [47], é a chave para melhorar sua produtividade e sua confiabilidade.

A atividade de teste de software tem sido considerada uma etapa importante no ciclo de desenvolvimento e pode chegar a consumir cerca de 50% do tempo e custo de desenvolvimento de um produto de software [16, 20, 44].

A idéia básica do teste de software consiste em executar um programa fornecendo dados de teste, necessários para a execução do programa, e comparar a saída alcançada com a saída esperada, obtida da especificação do software, com a finalidade de revelar o maior número possível de defeitos existentes, idealmente todos. O par formado com o dado de teste e sua sáida esperada forma um caso de teste.

Torna-se então necessária a distinção entre os conceitos de falha, defeito e erro. Segundo [18], uma falha é um evento notável em que o sistema viola a sua especificação, um defeito é uma deficiência algorítmica que pode levar a uma falha do sistema e um erro é um estado incorreto do conjunto de dados do programa.

Uma forma de maximizar o número de defeitos revelados seria a execução exaustiva, onde todos os dados do domínio de entrada seriam testados. Porém, se o domínio de entrada for infinito, a atividade de teste nunca terminaria e o custo operacional do teste seria muito grande [22, 44]. Por esses motivos, a escolha de casos de teste bem elaborados assume um importante papel para a produtividade do teste.

Além da elaboração de casos de teste, um outro fator importante na aplicação do teste é saber se um programa foi suficientemente testado. Para auxiliar na elaboração de bons casos de teste e ainda decidir se a etapa de teste pode ser encerrada, técnicas e metodologias utilizam-se de *critérios de teste.*

Os critérios de teste estabelecem o que deve ser testado no programa, formando o conjunto de elementos requeridos. Tendo conhecimento dos elementos requeridos a serem testados, pode-se guiar a escolha de dados que devem ser utilizados para testar tal conjunto com o objetivo de aumentar a produtividade do teste. Os dados de teste podem ser gerados utilizando informações sobre o código, especificação do programa, informações históricas de erros comuns no processo de desenvolvimento, ou ainda, podem ser gerados aleatoriamente [22].

Quando um conjunto de dados de teste satisfaz todas as condições estabelecidas por um critério, diz-se que tal conjunto satisfez o critério [45].

De uma forma geral, pode-se dizer que o teste envolve as atividades de planejamento, projeto de casos de teste, execução e avaliação dos resultados dos testes.

Segundo Myers [16], o objetivo do teste de software pode ser estabelecido através das seguintes regras:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro;
- Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto;
- Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

Uma estratégia de teste inclui quatro níveis de teste [44]. São eles:

- Teste de unidade - tem como função detectar defeitos no módulo, a menor unidade do software;
- Teste de integração - sua função é testar as relações e interfaces entre os módulos. É realizado após o término do teste de unidade de todos os módulos envolvidos na integração;
- Teste de Validação - tem a finalidade de identificar erros de função e/ou características de desempenho com base na análise de requisitos do sistema. É realizado após o teste de integração.

- Teste de Sistema - visa realizar teste no software interagindo com outros elementos do sistema. São projetados testes para testar informações que chegam de outros elementos do sistema [44].

Os métodos para testar software são baseados essencialmente em três técnicas de teste de software:

- Técnica Funcional;
- Técnica Estrutural;
- Técnica Baseada em Erros.

Essas técnicas de aplicação de teste não são alternativas e uma boa estratégia pode ser a combinação entre elas pois cada técnica possui uma classe diferente de critério de teste podendo assim, detectar classes diferentes de erros.

Nas seções a seguir, tais técnicas são apresentadas.

3.1 Técnica Funcional

Segundo Pressman [44], os produtos de engenharia podem ser testados conhecendo a função específica que o produto deve executar. Dessa forma, testes podem ser realizados para demonstrar que cada função é totalmente operacional. Esta abordagem, também chamada de teste de caixa preta (*black box*), procura examinar aspectos do sistema sem se preocupar com a implementação. O testador deriva conjuntos de condições de entrada que exercitem completamente todos os requisitos funcionais para um programa.

O teste de caixa preta pode ser utilizado para: testar as interfaces do software, podendo assim demonstrar que as funções do software são operacionais; demonstrar que a entrada é adequadamente aceita e produz a saída correta; demonstrar que a integridade das informações externas é mantida.

A técnica funcional permite identificar defeitos como: erros de interface, funções incorretas ou perdidas, erros na estrutura de dados ou no acesso externo à base de dados, erros de performance, erros de terminação ou inicialização.

Segundo [44], os critérios para a técnica funcional são:

- Particionamento em Classes de Equivalência - divide-se o domínio de entrada do programa em classes de dados, classes de equivalência. A classe representa um conjunto de estados válidos ou inválidos das condições de entrada, usualmente expressas em uma sentença ou frase de especificação. É assumido que o teste de um valor representativo de uma classe é equivalente a um teste de qualquer outro valor da classe. Neste critério os casos de teste são obtidos a partir das classes identificadas;

- Análise de Valor Limite - este critério parte da premissa de que os erros tendem a ocorrer nos limites das classes de equivalência, logo, os casos de teste são selecionados de forma a exercitarem as condições limites das classes de equivalência, tanto no domínio de entrada como no domínio de saída;

- Técnicas de Grafo de Causa-Efeito - técnica de projeto de casos de teste que oferece uma representação concisa das condições de entrada - causas, e das ações correspondentes - efeitos. Este critério estabelece requisitos baseados nas possíveis combinações das condições de entrada. Os passos para a aplicação do critério são:

  1. Dividir a especificação em módulos de tamanho e complexidade razoáveis;
  2. Identificar causas e efeitos na especificação;
  3. Construir o grafo de causa-efeito;
  4. Converter o grafo em uma tabela de decisão; e
  5. Converter as regras da tabela de decisão em casos de teste.

3.2 Técnica Estrutural

Nesta técnica, os casos de teste são gerados conhecendo o funcionamento interno do software - a implementação. Isso permite que testes sejam realizados para garantir que a operação interna do software tenha um desempenho de acordo com as especificações e que os componentes internos foram adequadamente postos à prova. Esta abordagem chama-se

teste de caixa branca (*white box*). O teste de caixa branca baseia-se no exame minucioso dos detalhes procedimentais, e a partir do conhecimento da estrutura interna do programa busca-se caracterizar um conjunto de componentes elementares a serem exercitados [22].

O teste estrutural utiliza uma representação do programa conhecida como grafo de fluxo de controle ou grafo de programa. Esta representação é um grafo orientado com um nó único de entrada e um nó único de saída. Cada nó do grafo representa um bloco de comandos indivisíveis, onde se o primeiro comando do bloco for executado, todos os outros também o serão em ordem e não existe desvio de execução para nenhum comando dentro do bloco. Os nós são ligados por arcos direcionados que indicam possíveis transferências de controle entre os blocos.

Um conceito importante para essa técnica é o *caminho*, que é definido como uma seqüência finita de nós interligados por arcos. Outro conceito é o conjunto de elementos requeridos, que são condições a serem satisfeitas durante o teste para que este seja realizado com sucesso. Os elementos podem ser: conjunto de nós ou de arcos do grafo de programa; determinados tipos de caminhos no programa; determinadas associações entre definição e uso de variáveis; entre outros.

Os critérios de teste estrutural são, em geral, classificados em:

- Critérios Baseados em Fluxo de Cotrole;
- Critérios Baseados em Fluxo de Dados;
- Critérios Baseados na Complexidade.

Visto que os critérios baseados em fluxo de controle e dados fazem parte do focu deste trabalho, estes serão detalhados nas próximas seções.

3.2.1 Critérios Baseados em Fluxo de Cotrole

Critérios desta clasee, utilizam características de controle da execução do programa para determinar quais elementos são requeridos. Os critérios mais conhecidos dessa classe são [43]:

- Todos-Nós - requer que a execução do programa passe pelo menos uma vez em cada nó do programa, fazendo com que todos os comandos do programa sejam executados;

- Todos-Arcos - exige que a execução exercite cada arco do grafo de programa pelo menos uma vez, fazendo com que todos os desvios de fluxo de controle do programa sejam executados;

- Todos-Caminhos - requer que todos os caminhos possíveis, derivados do grafo do programa sejam executados.

3.2.2 Critérios Baseados em Fluxo de Dados

Estes critérios utilizam informações de fluxo de dados para determinar os requisitos de teste. Exploram as interações que envolvem definições de variáveis e referências a essas definições. Os critérios mais conhecidos dessa classe são [45]:

- Todos-Usos - requer que todas as associações entre uma definição de variável e seus usos (computacional ou predicado) [1] subseqüentes sejam exercitados por ao menos um caminho onde a variável não seja redefinida, caminho livre de definição;

- Todos-Potenciais-Usos - requer que todas as associações entre uma definição de variável e seus potenciais[2] usos (computacional ou predicado) subseqüentes sejam exercitados por ao menos um caminho livre de definição;

- Todos-Potenciais-Usos/Du - requer que todas as associações entre uma definição de variável e seus potenciais[3] usos (computacional ou predicado) subseqüentes sejam exercitados por ao menos um caminho livre de definição e que seja livre de laço;

- Todos-Potenciais-Du-Caminhos - requer que todos os potenciais du-caminhos[4] sejam exercitados.

[1] Um uso computacional permite modificar ou observar o resultado de uma definição anterior(ex. atribuição). Um uso predicado afeta o fluxo de controle do programa(ex. if).

[2] Não exige um uso, basta ser um local onde possa existir um uso.

[3] Não exige um uso, basta ser um local onde possa existir um uso.

[4] Um potencial du-caminho é um caminho no grafo desde um nó i onde haja definição uma variável x, até um outro nó qualquer j livre de definição com relação a x e livre de laço.

3.3 Técnica Baseada em Erros

A técnica baseada em erros utiliza informações sobre as classes de erros mais frequentes no processo de desenvolvimento de software, fazendo com que os requisitos do teste sejam derivados dessas classes com o objetivo de mostrar a ausência ou a presença de erros nos programas. Os critérios típicos são Semeadura de Erros [49] e Análise de Mutantes [41].

3.3.1 Análise de Mutantes

Segundo a hipótese do programador competente apresentada por DeMillo [41], os programadores experientes fazem seus programas similares ao correto, de acordo com a especificação. Uma outra hipótese a ser considerada é o efeito de acoplamento. Este diz que erros complexos estão relacionados a erros simples, no sentido de que conjuntos de casos de teste capazes de identificar erros simples são também capazes de identificar os complexos.

Com base nessas duas hipóteses a análise de mutantes funciona da seguinte maneira:

- Dado um programa a ser testado P e um conjunto de casos de teste T;
- O programa P é executado com T;
- Se apresentar resultados incorretos, então o mutante é considerado morto, seu erro é encontrado e o teste termina;
- Caso contrário, o programa pode ainda conter erros que o conjunto T não foi capaz de revelar e o teste continua. Erros comumentes praticados são introduzidos através de modificações sintáticas simples, de forma a não causar um erro sintático. Essas modificações alteram a semântica do programa P produzindo programas $P1$, $P2$, ..., Pn, denominados mutantes de P;
- Supondo que o programa original P esteja correto, os programas mutantes devem ter comportamento incorreto, já que sofreram modificações sintáticas. Casos de teste devem ser elaborados para mostrar que tais modificações levam a um programa incorreto;

- Os programas mutantes devem ser executados com o mesmo conjunto de teste T com o objetivo de obter apenas mutantes mortos e equivalentes. Um mutante é tido como morto se para algum caso de teste o resultado do mutante difere do programa original P. Um mutante é dito equivalente se apresenta sempre o mesmo resultado que o programa original P. A decisão sobre mutantes equivalentes requer intervenção do testador.

Os erros sintáticos simples são introduzidos através de regras que definem as alterações a serem aplicadas no programa original P. Essas regras são chamadas de operadores de mutação.

Os operadores são utilizados para ou introduzir mudanças sintáticas simples, com base nos erros típicos cometidos pelos programadores (como trocar o nome de uma variável), ou forçar determinados objetivos de teste (como executar cada arco do programa) [41].

3.4 Ferramentas de Teste

Entre as ferramentas para automatizar a etapa de teste de software destacam-se: ASSET [12], ATAC [17], POKE-TOOL [35] e PROTEUM [34]. Neste trabalho serão tratadas com mais detalhe as Ferramentas POKE-TOOL E PROTEUM.

3.4.1 Ferramenta POKE-TOOL

A POKE-TOOL é uma ferramenta que apóia os critérios de teste da Família de Critérios Potenciais Usos (FCPU) [22, 35] para o teste de unidade. A ferramenta foi desenvolvida, a princípio, para atender programas escritos em C, apesar de possuir versões para atender linguagens como Pascal e Ada e poder ser configurada para atender outras linguagens [36]. A aplicação da ferramenta é orientada para a sessão de trabalho, onde o usuário/testador pode realizar as seguintes atividades de teste: análise estática da unidade a ser testada; preparação para o teste; submissão de casos de teste; avaliação de casos de teste e gerenciamento dos resultados de teste. Para realizar essas atividades, a ferramenta executa duas fases, a fase estática e a fase dinâmica.

A fase estática consiste em analisar o código fonte do programa a ser testado para obter todas as informações necessárias para a aplicação dos critérios de teste. É nessa fase também que é realizada a instrumentação do código fonte, inserindo pontas de prova e gerando uma nova versão da unidade em teste, a versão instrumentada. Concluída essa fase, o usuário pode solicitar informações e, com base nessas, projetar seus casos de teste.

A fase dinâmica consiste do processo de execução e avaliação de casos de teste. Para isso, é necessário que seja gerado o programa executável da versão instrumentada. Nesta fase, a ferramenta fornece ainda, o conjunto de caminhos ou associações que foram executados, as entradas, as saídas e os caminhos percorridos para cada caso de teste.

A ferramenta é composta por vários módulos, à saber: poketool; li; chanomat; pokernel; gera executável; executa caso de teste e avaliador. Tais módulos implementam as funções das fases estática e dinâmica e se comunicam através de arquivos.

Os módulos que tratam da fase estática da ferramenta são: li, chanomat e pokernel e os módulos referentes à fase dinâmica do funcionamento da ferramenta são: gera executável, executa caso de teste e avaliador.

A seguir será brevemente descrito cada um desses módulos que compõem a Ferramenta POKE-TOOL, conforme [35].

- Módulo poketool: responsável pela interface com o testador, fornecendo todos os menus, pedindo todas as informações necessárias para o teste e permitindo ao usuário solicitar informações sobre o teste. Esse módulo coordena a atividade dos outros módulos;

- Módulo li: este módulo toma o código fonte do programa em teste como entrada e faz a tradução para uma linguagem intermediária (LI), gerando um arquivo '.li' como saída. Essa LI gerada é uma representação do programa em teste e é sobre ela que o teste é realizado;

- Módulo chanomat: toma como entrada o arquivo resultante do módulo li, calcula o grafo de fluxo de controle e modifica o arquivo que contém o programa traduzido para LI, incluindo mais uma marcação que diz em qual nó do grafo de programa se

encontram os comandos traduzidos. As saídas desse módulo são o arquivo com a nova LI, arquivo '.nli' e um arquivo que descreve o grafo de fluxo de controle;

- Módulo pokernel: é o responsável pelo restante da análise estática da unidade em teste, gerando as informações estáticas adicionais necessárias para o teste dinâmico da unidade. Toma como entrada o arquivo '.nli' gerado pelo módulo chanomat. Suas principais funções são: cálculo dos arcos primitivos; extensão do grafo de fluxo de controle, instrumentação, construção do grafo(i) e geração dos descritores;

- Módulo gera executável: fornece condições para a geração do programa executável da versão instrumentada. Tem como função selecionar o compilador e como saída o programa executável da versão instrumentada;

- Módulo executa caso de teste: controla a execução dos casos de teste salvando as entradas, a saída e o caminho executado para cada caso de teste. Tem como entrada o código-executável e os casos de testes. Tem como saída os caminhos executados pelos casos de testes, as saídas obtidas e a entrada dos casos de testes;

- Módulo avaliador: identifica os caminhos ou associações que são executados pelos casos de teste e fornece uma análise da cobertura do conjunto de casos de teste fornecido.

Os módulos são integrados através de "scripts shell" [36] que podem ser acionados por linha de comando ou por uma interface gráfica.

O script poketool, gera o programa instrumentado, os elementos requeridos, informações estáticas do programa e o grafo do fluxo de controle do programa em teste.

O script pokeexec executa o programa instrumentado direcionando para arquivos informações como os dados de entrada, os parâmetros fornecidos, as saídas obtidas e os caminhos executados para cada caso de teste.

O script pokeaval avalia a cobertura atingida com o conjunto de casos de teste aplicados, fornece os elementos executados e os não executados.

3.4.2 Ferramenta PROTEUM

A Ferramenta PROTEUM (PROgram Testing Using Mutants) foi desenvolvida no ICMC-USP para apoiar o critério Análise de Mutantes [41] em programas escritos na linguagem C, apesar de poder ser configurada para ser aplicada sobre qualquer linguagem de programação procedimental [34].

Semelhantemente à POKE-TOOL, a Proteum também utiliza-se da sessão de teste. A sessão possibilita ao usuário/testador criar um teste de programa, interrompê-lo e retomá-lo mais tarde. A ferramenta permite a realização de tarefas de teste como: definição de conjunto de casos de teste, execução do programa em teste, seleção dos operadores de mutação, geração de mutantes com os operadores de mutação, execução dos mutantes com o conjunto de casos de teste, análise dos mutantes vivos e cálculo do escore de mutação.

Os dados de teste podem ser fornecidos de duas maneiras para a ferramenta: interativamente e a partir de arquivo. Interativamente, o programa é executado e o usuário fornecendo as entradas, assim pode verificar a saída produzida e comparar com a saída esperada. Caso não esteja de acordo, um erro já foi encontrado e o programa deve ser corrigido. Caso esteja de acordo, as entradas e saídas são armazenadas formando os casos de teste. No caso de serem fornecidas entradas através de arquivos, a PROTEUM utiliza os dados do arquivo como entrada para o programa, executa-os e armazena as saídas formando os casos de teste.

Para a geração dos mutantes, a ferramenta conta com 71 operadores de mutação divididos em quatro classes: mutação de comando (*statement mutations*), mutação de operadores (*operator mutations*), mutação de variáveis (*variable mutations*) e mutação de constantes (*constant mutations*) [34]. Os operadores podem ser escolhidos de acordo com a classe de erros que se deseja encontrar [34] e para cada operador, o usuário pode especificar um percentual de mutantes a serem gerados.

Para realizar a avaliação do conjunto de teste, a ferramenta gera, compila e executa cada mutante e compara seu comportamento com o apresentado pelo programa original. A execução pode ser realizada com dados suficientes para matar o mutante (execução em

modo normal), ou pode ser realizada sobre todos os dados do conjunto de teste independente de o mutante já ter sido morto (execução em modo research).

Depois da execução, a ferramenta apresenta o status do teste com informações como o número total de mutantes, o número de mutantes vivos, o escore de mutação, entre outras.

O testador pode então analisar os mutantes ainda vivos para identificar os mutantes equivalentes e os que não podem ser mortos. Para melhorar a qualidade dos dados de teste, o usuário pode ainda inserir novos casos de teste. Este processo pode ser repetido até que um escore satisfatório seja alcançado.

3.5 Considerações Finais

Neste capítulo foram apresentados os principais critérios de teste existentes e ferramentas de teste que apoiam a utilização desses critérios. Destacam-se as Ferramentas POKE-TOOL e PROTEUM que implementam respectivamente critérios baseados em fluxo de controle e fluxo de dados, e o critério Análise de Mutantes, baseado em erros.

As ferramentas aqui descritas e a maioria das existentes, não realizam a geração automática de dados de teste, devido a algumas dificuldades inerentes à própria atividade de teste, mencionadas na Introdução (Capítulo 1) desta dissertação.

Por ser de especial interesse para o presente trabalho, a tarefa de geração de dados de teste para satisfazer critérios de teste será abordada separadamente no próximo capítulo.

CAPÍTULO 4

GERAÇÃO DE DADOS DE TESTE

Um dado critério de teste requer um conjunto de elementos a serem exercitados pelos casos de teste e fornece uma medida de cobertura baseada no número de elementos exercitados ou cobertos.

A tarefa de geração de dados para satisfazer um determinado critério diz respeito à geração de dados de entrada para o programa em teste capaz de exercitar todos os elementos requeridos. Esta é uma das etapas mais importantes, porém mais difícil de ser automatizada devido à algumas limitações inerentes à própria atividade de teste [22]:

- não existe algoritmo de propósito geral para determinar um conjunto de casos de teste T que satisfaça um determinado critério de teste C, nem mesmo para saber se T existe [45].
- não existe procedimento para comprovar a corretude de um programa;
- é indecidível se dois programas computam a mesma saída, ou seja, se eles são equivalentes;
- é indecidível saber se um determinado caminho é ou não executável;
- é indecidível detectar a ocorrência de correção coincidente onde um item incorreto, coincidentemente produz resultado correto.

4.1 Técnicas de Geração de Dados de Teste

Diversas técnicas para geração automática de dados de teste podem ser encontradas atualmente. Entre as mais comuns estão:

- Geração de Dados de Teste Aleatória;

- Geração de Dados de Teste Baseada em Execução Simbólica; e
- Geração de Dados de Teste Baseada em Execução Dinâmica.

Essas técnicas têm sido aplicadas com diferentes graus de sucesso, devido às limitações da atividade de teste mencionadas acima. Para lidar com essa situação, alguns autores propuseram o uso de algoritmos meta-heurísticos tais como AG, Busca Tabu, etc, originando um novo campo de pesquisa chamado *Teste Evolucionário* [21].

4.1.1 Geração de Dados de Teste Aleatória

A técnica de geração de dados aleatória consiste em gerar dados que servirão de entrada para o programa até que uma entrada útil seja encontrada [14]. O problema dessa técnica encontra-se na geração de dados de teste para programas complexos ou para satisfazer um critério complexo, onde uma entrada pode ter que satisfazer requisitos muito específicos, podendo assim, a técnica produzir poucas entradas adequadas se comparado ao domínio de entrada [14]. Dessa forma a probabilidade de uma entrada adequada ser selecionada pode ser muito pequena. Em contrapartida é um método bastante utilizado, principalmente para comparação, devido a sua fácil aplicação.

4.1.2 Geração de Dados de Teste Baseada em Execução Simbólica

Esta técnica consiste em atribuir valores simbólicos às variáveis para através de uma abstração obter uma caracterização matemática de o que o programa faz. Dessa forma a geração de dados de teste se reduz ao problema de resolver uma expressão algébrica [30]. Alguns problemas na utilização desta técnica surgem com laços indefinidos onde pode ser necessário verificar o comportamento se o laço não é executado, é executado uma vez, duas e assim por diante. Outro problema ocorre quando se realiza uma referência indireta para uma variável (por exemplo, acesso em vetor).

4.1.3 Geração de Dados de Teste Baseada em Execução Dinâmica

Esta técnica de geração de dados baseia-se na idéia de que partes do programa podem ser vistas como funções. O programa é executado até que certo ponto seja alcançado, os valores das variáveis são gravados e tratados como se fossem valores de uma função. Dessa forma, quando é pretendido que a execução tome um rumo, basta resolver uma função de minimização formada pelas variáveis que fazem parte da condição do desvio [50].

4.1.4 Teste Evolucionário

Segundo Wegener [21], trabalhos sobre teste evolucionário podem ser divididos em duas categorias com base na forma em que a função de *fitness* é avaliada:

- uma primeira categoria onde a função de *fitness* de cada indivíduo é calculada analisando uma medida de cobertura do caso de teste correspondente [2, 33, 37, 39, 42, 46], e

- uma segunda categoria onde a função de *fitness* é orientada por uma meta de teste [7, 13, 15], ou seja, leva em consideração que AG é uma opção para problemas de otimização e que a utilização da técnica dinâmica para gerar dados de teste para o teste estrutural pode ser transformada em um problema de otimização [38].

Entre os trabalhos que empregam AG para geração de dados destacam-se os de Jones et al [5], Roper et al [33], Michael et al [14].

No trabalho desenvolvido por Jones et al, a técnica AG é utilizada na geração de dados de teste para o critério estrutural Todos-Arcos. As variáveis de entrada são mapeadas por codificação e concatenação, formando um string de bits. A função de *fitness* é baseada no predicado associado a cada ramo e no número de iterações de laço requeridas. O algoritmo genético implementa a seleção aleatória, recombinação aleatória e mutação com probabilidade dependente do comprimento do string. Como resultado, este trabalho revelou que o conjunto de dados requereu uma quantidade de dados menor para satisfazer o critério em relação à geração aleatória e à boa qualidade dos dados gerados avaliados utilizando análise de mutantes.

O trabalho de Roper et al [35], visa gerar dados de teste para atingir um determinado nível de cobertura do código do programa utilizando também o critério Todos-Arcos. Os indivíduos na população codificam os valores de entrada das variáveis do programa a ser coberto. A função de *fitness* é baseada na cobertura que o indivíduo obteve no código do programa e a população inicial é gerada aleatoriamente.

O trabalho desenvolvido por Michael et al [7], objetiva 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. O critério utilizado é também Todos-Arcos. Para cada condição atingida, é realizada uma minimização baseada em Algoritmos Genéticos na tentativa de executar um caminho ainda não executado. A população inicial do Algoritmo Genético contém os testes que atingem a condição mas caso a condição não tenha sido atingida suficientemente pode conter ainda valores aleatórios. Como resultado foi constatado que para um mesmo número de execuções o algoritmo de busca com Algoritmo Genético atingiu 60% das condições, enquanto a técnica de geração de dados de teste aleatória conseguiu apenas 41%. No mesmo trabalho foi constatado também que conforme a complexidade do problema aumenta, aumenta a eficiência do primeiro sobre o segundo.

No trabalho de Bueno [38], é proposta uma ferramenta utilizando a técnica dinâmica para geração de dados de teste e Algoritmos Genéticos para otimização, fazendo manipulações nos valores de entrada tentando executar um dado caminho pretendido. Diferentemente dos trabalhos citados, este visa a satisfazer os critérios baseados em fluxo de dados e família de Potenciais Usos. Os indivíduos da população representam um dado de entrada para o programa de forma codificada. São utilizados conceitos de algoritmos genéticos simples como seleção, crossover e mutação. A função de *fitness* utiliza informações como número de nós coincidentes entre o caminho executado pelo indivíduo e o caminho desejado, o módulo de uma função de predicado onde houve o desvio do caminho pretendido e o maior valor para a função de predicado apurado dentre os caminhos que executaram o mesmo número de nós corretos. Em resumo, a função de *fitness* representa o quanto um indivíduo está próximo do caminho desejado em função do número de nós

corretos executados.

4.2 Considerações Finais

Neste capítulo foram descritas as principais técnicas de geração de dados de teste e os principais trabalhos de teste evolucionário.

A maioria dos trabalhos existentes têm como objetivo a satisfação do critério Todos-Arcos e abordam um critério específico. Esses trabalhos também não oferecem um ambiente integrado com diferentes ferramentas de teste.

Os trabalhos na segunda categoria, função de *fitness* direcionada por meta de teste, foram propostos com o objetivo de melhorar a cobertura alcançada por um conjunto de teste executando um particular elemento requerido. Porém, o uso de algumas estratégias e de hibridização nos AG's pode melhorar o desempenho dos trabalhos da primeira categoria, função de *fitness* baseada na cobertura dos indivíduos.

O Framework *TDSGen* visa a resolver essas duas questões. Explorar o uso de AG híbridos e oferecer um ambiente para aplicação dos critérios, interagindo com duas ferramentas de teste.

CAPÍTULO 5

FERRAMENTA *TDSGEN*

5.1 Descrição

A Ferramenta *TDSGen* foi implementada em C++ para plataforma Linux. Seu nome foi originado das iniciais de **T**est **D**ata **S**et **Gen**erator[1] - *TDSGen*. Esta ferramenta integra as Ferramentas de teste POKE-TOOL e PROTEUM, descritas no Capítulo 3, com a geração de dados de teste baseada em AG's com algumas modificações e com a implementação de algumas estratégias. Esta integração possibilita a aplicação dos critérios de teste implementados pelas ferramentas na geração de dados de teste. Os critérios são:

- Todos-Nós;
- Todos-Arcos;
- Todos-Potenciais-Usos;
- Todos-Potenciais-Du-Caminhos;
- Todos-Potenciais-Usos/Du;
- Análise de Mutantes;

Exceto pelo critério Análise de Mutantes, que é implementado pela Ferramenta PROTEUM, os outros critérios são todos apoiados pela POKE-TOOL.

A estrutura de integração da Ferramenta *TDSGen* com as Ferramentas POKE-TOOL e PROTEUM pode ser melhor visualizada na Figura 5.1.

Este trabalho resultou no desenvolvimento de um ambiente de teste que apóia a aplicação completa de estratégias de teste para critérios de teste das técnicas estruturais e

[1]Gerador de Conjunto de Dados de Teste

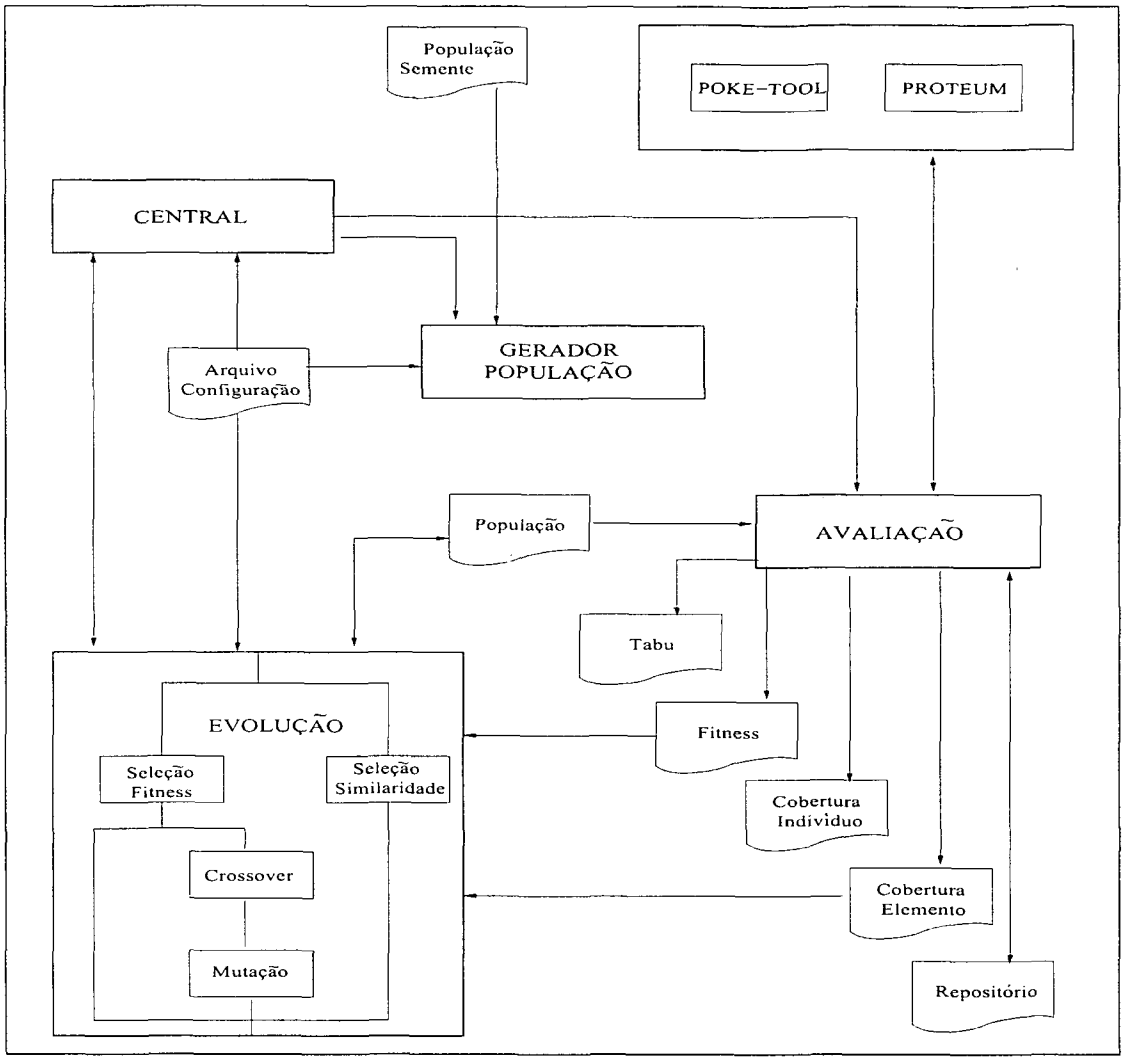

Figura 5.1: Estrutura de integração da ferramenta

baseadas em erros, diferenciando o mesmo de trabalhos presentes na literatura que proporcionam um ambiente de teste focados em um critério somente, geralmente critérios estruturais [11].

A ferramenta permite ainda outras vantagens tais como: uso de dados de teste iniciais derivados de técnicas funcionais como dados resultantes da aplicação de abordagens complementares como outros critérios, não necessita que o testador analise o código do programa, permite testar uma variedade de programas escritos em C (pois possui um esquema flexível de codificação da entrada de dados para programas em teste) e permite que os dados de teste gerados sejam utilizados nas Ferramentas POKE-TOOL e PROTEUM.

As Ferramentas de teste, POKE-TOOL e PROTEUM, são utilizadas para avaliar os dados, proporcionando informações utilizadas para o cálculo da função de *fitness*. A configuração da ferramenta é toda baseada em arquivo, evitando interação do testador. Este arquivo esta melhor detalhado na Seção 5.1.1.

O AG implementado na ferramenta possui modificações incrementais (hibridização) para melhor atender a geração de um conjunto de dados de teste para os critérios acima citados, adicionando recursos extras configuráveis. Os recursos disponíveis para a customização da ferramenta são:

- lista Tabu - para manter apenas indivíduos com desempenho complementares;
- repositório de execuções - memorizando indivíduos já avaliados evitando reavaliação;
- parâmetros da ferramenta - restringindo tipos de dados;
- parâmetros do programa em teste - entrada de dados;
- parâmetros do AG - tamanho de população, número de gerações e taxas usadas no processo de evolução.

As próximas seções abordam esses recursos com mais detalhes além de outros aspectos principais sobre a ferramenta. O fluxo de execução da ferramenta pode ser observado nas figuras apresentadas no Apêndice A.

5.1.1 Interpretação do Arquivo de Configuração

Esta etapa de execução consiste em interpretar o arquivo de configuração para obter os parâmetros necessários para a execução da ferramenta. Os parâmetros passados neste arquivo devem obedecer as seguintes regras:

- um parâmetro por linha;
- cada declaração deve estar na forma "parâmetro = valor;" - onde pârametro é precedido de #, e
- qualquer linha que não siga este padrão é ignorada.

Os parâmetros aceitos são apresentados na Tabela 5.1.

Tabela 5.1: Parâmetros do arquivo de configuração

Parâmetros	
#ArquivoFonte	#TaxaCrossover
#NomeFuncao	#Elitismo
#Criterio	#Similaridade
#CoberturaCriterio	#*Fitness*
#FormatoEntrada	#AtivaTabu
#NumeroArgumentos	#VariacaoInt
#ArquivoPopulacao	#Variacao*string*
#TamanhoPopulacao	#Tipo*string*
#NumeroGeracoes	#PausaGeracao
#TaxaMutacao	#GeracoesComRepositorio

Exemplo de uma linha de configuração para informar o nome do programa a ser testado:

#ArquivoFonte = compress.c;

5.1.2 Configuração do Ambiente de Teste

Tanto a Ferramenta POKE-TOOL como a PROTEUM, iniciam o processo de teste com uma sessão. Como a *TDSGen* utiliza essas ferramentas para obter informações de elementos requeridos e avaliação de dados de teste, uma sessão deve ser inicialmente criada.

Além da sessão, algumas restrições como diretório para o fonte do programa e variáveis de ambiente devem ser devidamente ajustados para utilizar as ferramentas.

5.1.3 Obtenção da População

A população inicial consiste em obter dados de teste para o programa em análise e codificá-los em forma de indivíduos, para uso do AG. Este processo pode ser realizado de duas maneiras:

- Arquivo de população inicial - Um arquivo texto contendo um dado de teste por linha, com valores separados por vírgula pode ser fornecido para servir como uma

semente para população inicial do AG. O conteúdo do arquivo deve refletir o arquivo de configuração como por exemplo quantidade de entradas e seus tipos de dados;

- Aleatoriamente - Com base no arquivo de configurações, que informa o tipo de dado com o qual o programa trabalha, um método de geração aleatória de dados foi desenvolvido para cada tipo suportado pela ferramenta, conforme listado na Tabela 5.2.

Após a obtenção dos dados de teste, estes são codificados conforme a abordagem da Seção 5.1.3.1, e depois são armazenados em arquivo.

5.1.3.1 Codificação de Indivíduos

A codificação obedece um critério de representação em bloco de tipo de dado. Cada tipo de dado possui uma representação específica de bloco. A Figura 5.2 ilustra a representação desses blocos conforme o tipo. Assim, se o programa possui entrada do tipo inteiro e *string*, a representação dos valores de entrada em indíduos será formada pela concatenação dos blocos correspondentes para os tipos inteiro e *string*.

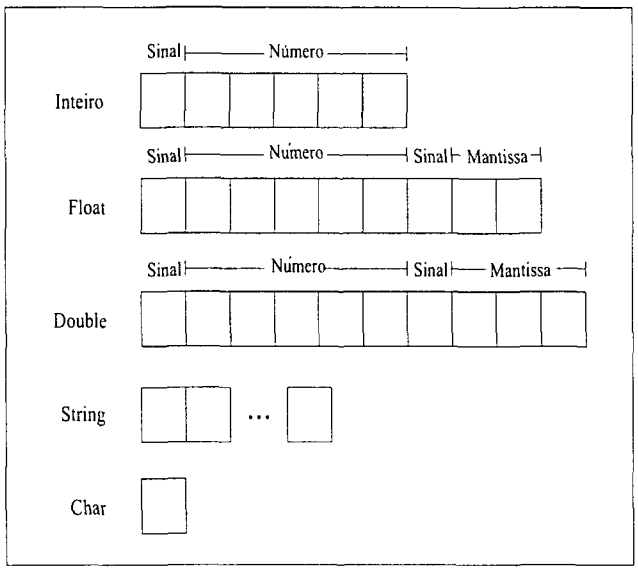

Figura 5.2: Ilustração da codificação dos tipos de dados

Para inteiro, utiliza-se uma máscara de seis posições. Uma posição para o sinal e cinco

para o número resultando em +00000.

Para *float*, utiliza-se uma máscara de nove posições. Uma posição para o sinal, cinco para o número, uma posição para o sinal da mantissa e duas posições para o valor da mantissa, resultando em +00000+00.

A máscara para o tipo *double* é semelhante à máscara de *float*, porém possui uma capacidade maior refletindo em mais uma posição para a mantissa, resultando em +00000+000.

Para o tipo *string*, a máscara consiste em preencher o conteúdo com um caracter coringa Ø. Para char, apenas um é necessário.

A Tabela 5.2 ilustra como é feita a codificação:

Tabela 5.2: Codificação de tipos de dados

Tipo	Máscara	Valor	Codificado
Inteiro	+00000	-543	-00543
Float	+00000+00	-543000	-00543+03
Double	+00000+000	-543000	-00543+003
String (tam 10)	ØØØØØØØØØØ	teste	testeØØØØØ
Char	Ø	A	A

Em termos práticos, se um programa necessita de dados dos tipos inteiro, *string* e char com os valores respectivamente, -543, "teste" e 'A', o indivíduo teria a seguinte representação [2]: -00543testeØØØØØA

5.1.4 Avaliação da População

A avaliação dos indivíduos consiste em gerar 1) um escore de *fitness*, com base em quantos elementos requeridos cada indivíduo satisfaz pelo total de elementos requeridos e 2) um escore de similaridade, que é um bônus que melhor bonifica indivíduos que satisfazem elementos inéditos.

Essa avaliação é feita através de interface com a ferramenta de teste que apóia o critério de teste configurado no arquivo de configuração. A ferramenta de teste gera em arquivo todos os elementos requeridos satisfeitos pela execução. Realizando uma nova

[2] O tamanho adotado para o tipo *string* neste exemplo foi 10. Este valor é informado no Arquivo de Configuração

execução para cada indivíduo da população, obtém-se o número de elementos satisfeitos por cada indivíduo da população, inclusive quais. Essas informações de cobertura são armazenadas em um arquivo de forma a representar uma tabela de cobertura de elementos por indivíduo, onde nas linhas estão representados os indivíduos e nas colunas os elementos requeridos. Para cada indivíduo, tem-se uma linha de cobertura onde o n-ésimo caracter dessa linha tem o valor de 'X' se o indivíduo satisfez o n-ésimo elemento requerido, '–' caso contrário. Este arquivo pode ser observado na Figura 5.3.

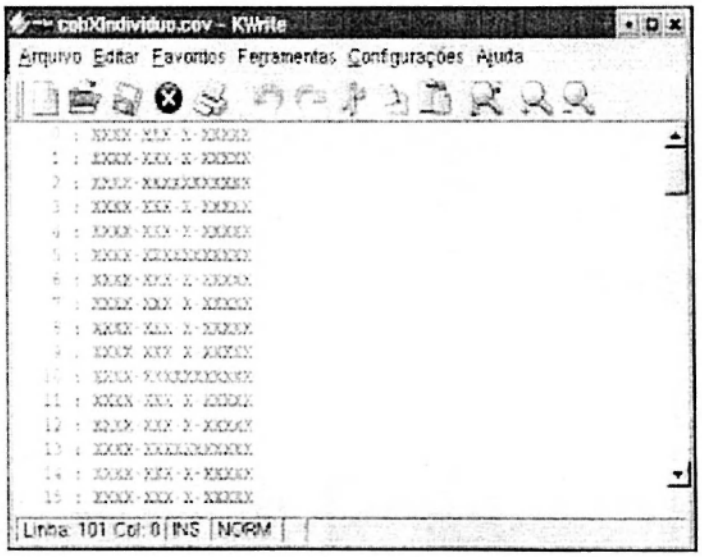

Figura 5.3: Ilustração do arquivo de cobertura dos indivíduos

A partir deste primeiro arquivo, calcula-se o escore de *fitness* para cada indivíduo com a Fórmula 5.1:

$$Fitness_x = \frac{nro\_elementos\_satisfeitos_x * 100}{nro\_total\_elementos} \tag{5.1}$$

Na fórmula 5.1, $nro\_elementos\_satisfeitos_x$ é o número de elementos requeridos satisfeitos pelo indivíduo sendo analisado e $nro\_total\_elementos$ é a quantidade total de elementos requeridos pelo critério de teste em questão, obtido com a ferramenta de teste.

Um outro arquivo também é gerado representando assim a relação de indivíduos que satisfazem um determinado elemento requerido. Neste arquivo as linhas representam os elementos requeridos e as colunas os indivíduos da população. Este segundo arquivo representa a transposta da matriz representada no primeiro arquivo. Para cada elemento

tem-se uma linha de cobertura onde o n-ésimo caracter dessa linha tem o valor de 'X' se o n-ésimo indivíduo satisfez o elemento requerido em questão, '–' caso contrário. Este arquivo é ilustrado na Figura 5.4.

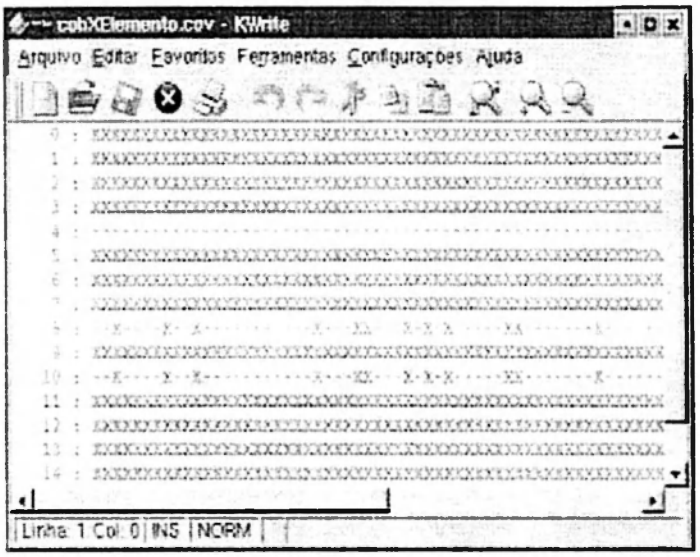

Figura 5.4: Ilustração do arquivo de cobertura dos elementos

A partir deste segundo arquivo, pode-se obter o total de indivíduos e quais indivíduos satisfizeram cada elemento, assim pode-se calcular um bônus de similaridade referente a um determinado elemento, com base no número de indivíduos que o satisfizeram. O escore final de similaridade de cada indivíduo é a somatória de todos os bônus que ele ganhou. Assim, um indivíduo que satisfaz poucos elementos, mas que apenas ele satisfaz, não é punido por ter um escore de *fitness* baixo no momento da evolução. A Fórmula 5.2 a seguir ilustra o cálculo da bonificação para o elemento requerido x.

$$BonusSimilaridadeElemento_x = 100 * \left(1 - \frac{nro\_individuos\_cobrem_x}{nro\_total\_individuos}\right) \tag{5.2}$$

Onde $nro\_individuos\_cobrem_x$ representa o número de indivíduos que cobrem o elemento em questão e $nro\_total\_individuos$ o número total de indivíduos, o tamanho da população.

O cálculo do escore de similaridade do indivíduo é ilustrado pela Fórmula 5.3:

$$BonusSimilaridadeIndividuo_y = \sum_{i=1}^{q} BonusSimilaridadeElemento_x \quad (5.3)$$

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

Como citado na Seção 5.1.3, os indivíduos são armazenados em arquivo, logo para se utilizar as ferramentas de teste, antes os indivíduos devem ser recuperados do arquivo de população, decodificados e depois utilizados pela ferramenta de teste.

O valor do escore de *fitness* de cada indivíduo é armazenado em outro arquivo, ficando disponível para a etapa de evolução detalhada adiante.

5.1.5 Evolução da População

A etapa de Evolução da População compreende em escolher indivíduos da população atual para gerar indivíduos que formarão a próxima população da nova geração. A ferramenta implementa diferentes estratégias de evolução:

- evolução por *Fitness*;
- evolução por Elitismo;
- evolução por Similaridade;

Além das diferentes estratégias de evolução, a ferramenta implementa ainda um mecanismo de memorização com base em lista Tabu que pode ou não ser usado, dependendo da configuração fornecida. Caso esteja configurado para usar este mecanismo, após a execução das estratégias de evolução citadas acima, é realizada a atualização da lista Tabu. Todas essas estratégias estão detalhadas nas seções subsequentes. A quantidade de indivíduos gerada na nova população por cada uma dessas estratégias é um dos parâmetros do arquivo de configuração.

5.1.5.1 Evolução por *Fitness*

A estratégia de evolução por *fitness* consiste no método tradicional de evolução de AG's, onde a evolução é baseada no escore de *fitness* dos indivíduos, ou seja, utiliza-se de um método de seleção para obter um par de indivíduos e depois aplicam-se os operadores genéticos para gerar dois indivíduos da próxima geração. Esse processo se repete até que uma nova população com mesmo tamanho seja gerada.

O método de seleção escolhido foi seleção por roleta. Este método foi implementado utilizando uma variável de controle que acumula a somatória do valor do escore de *fitness* de cada indivíduo, obtido do arquivo de *fitness* resultante da avaliação da população. Um valor é obtido aleatoriamente entre zero e o valor da somatória. O indivíduo selecionado é aquele que ao somar o seu valor de escore de *fitness* na variável de somatória, supera ou se iguala ao valor sorteado.

Os operadores genéticos implementados foram crossover e mutação. O crossover entre dois indivíduos ocorre entre os blocos e não sobre a representação como um todo, como pode ser observado na Figura 5.5. Isto permite que o operador genético seja diferente para cada tipo de dado e respeite as particularidades dos valores contidos nos blocos de dados quem formam o indivíduo, evitando assim, que valores inválidos sejam gerados caso o mesmo fosse genérico. Para ilustrar o problema, se o operador de crossover fosse aplicado na décima quinta posição dos indivíduos da Figura 5.5, um valor inválido seria gerado. Primeiro verifica-se para cada bloco da representação dos indivíduos se ocorrerá ou não crossover, e caso sim, o operador crossover de um ponto, explicado no Capítulo 2, é aplicado entre os blocos. A mutação ocorre sobre os dois indivíduos resultantes do operador crossover, independente se foi ou não aplicado crossover, da mesma forma como explicada no Capítulo 2 e da mesma forma como o operador crossover, é aplicado sobre cada bloco da representação do indivíduo.

Se o resultado da aplicação dos operadores genéticos gerar um indivíduo que já faz parte da nova população, este é descartado (eliminação de duplicata abordada no Capítulo 2) e o processo continua até que o número de indivíduos configurados para serem gerados por este método de evolução seja alcançado.

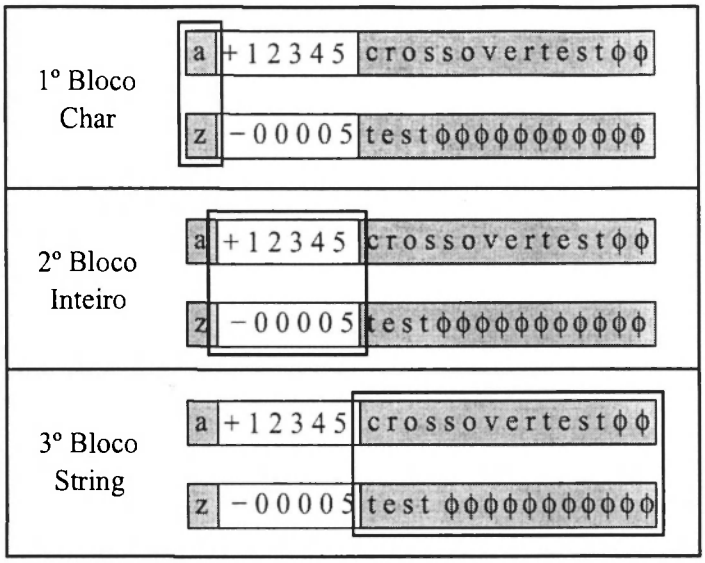

Figura 5.5: Aplicação dos operadores genéticos

5.1.5.2 Evolução por Elitismo

Esta estratégia introduz indivíduos na próxima geração com base em uma lista ordenada pelo escore de *fitness* de forma a garantir que bons indivíduos não sejam perdidos no processo de evolução por *fitness*. Caso o indivíduo já exista na nova população, o próximo da fila é selecionado e assim sucessivamente, eliminando a duplicidade de indivíduos conforme abordado no Capítulo 2.

5.1.5.3 Evolução por Similaridade

Esta estratégia é similar à evolução por elitismo porém ao invés de ordenar a lista pelo escore de *fitness*, ordena pelo valor de similaridade gerado na avaliação da população. Esta estratégia busca garantir que indivíduos que satisfaçam elementos inéditos não sejam perdidos durante o processo de evolução.

5.1.5.4 Estratégia Lista Tabu

A estratégia Lista Tabu implementada é uma lista onde um indivíduo é adicionado caso ele satisfaça a restrição da lista que consiste em manter indivíduos com cobertura

complementar. Ou seja, um indivíduo x é adicionado na lista se e somente se nenhum outro indivíduo da lista cobre algum elemento que x cobre ou se x possui uma abrangência de cobertura que melhora a cobertura de um indivíduo da lista, caso em que o último indivíduo deve ser removido da lista Tabu para a inserção de x. O tamanho da lista Tabu é limitado ao número de elementos requeridos obtidos, caracterizando a situação em que um indivíduo seria necessário para satisfazer cada um dos elementos requeridos.

Dessa forma a lista Tabu assume um papel importante para melhorar a cobertura da população, consequentemente do critério, por fornecer um método de memorização de indivíduos eficientes por todo o processo de evolução do AG.

5.1.6 Resultado

O resultado gerado pela ferramenta é um arquivo (chamado "Populacao.res"), contendo os dados de teste correspondentes à população que obteve melhor cobertura durante a execução da ferramenta, para o critério configurado. Esta população possui tamanho conforme determinado no arquivo de configuração e formato conforme as Ferramentas POKE-TOOL e PROTEUM podem utilizar como entrada de dados.

Caso a ferramenta esteja configurada para utilizar o recurso da lista Tabu, parâmetro "#AtivaTabu = sim;", um outro arquivo é gerado, "Tabu.res", contendo apenas os indivíduos mais eficientes de todo o processo. O resumo da cobertura de todas as gerações é gerado em um arquivo chamado "resumo.tst" e um relatório da execução é gerado em um arquivo chamado "resultado.tst" informando dados sobre a cobertura inicial, tempo para a avaliação inicial, a melhor cobertura e sua geração de descoberta e o tempo de execução final.

5.2 Configuração da Ferramenta

A utilização da ferramenta inicia com a geração do arquivo de configuração. Este consiste em um arquivo texto contendo palavras chaves seguidas de valores determinando o comportamento da ferramenta. O arquivo de configuração permite ao testador fornecer

informações que podem ser divididas em quatro categorias:

- relacionados com a atividade de teste de software: arquivo fonte do programa em teste, nome da função, critério de teste, cobertura desejável;
- parâmetros relacionados com o processo de evolução: taxa de crossover, taxa de mutação, tamanho da população, número máximo de gerações e informações sobre o processo de evolução - número de indivíduos evoluídos pelo processo tradicional de AG baseado em *fitness*, por elitismo ou por similaridade;
- relacionados com a execução do programa em teste: formato da entrada esperada pelo programa, número de argumentos e nome do arquivo de população inicial;
- relacionados com os recursos disponíveis pela ferramenta: ativação da lista Tabu, número de gerações utilizando o recurso de repositório de execuções, pausa entre gerações, variação do tipo inteiro e *string*.

A Tabela 5.3 apresenta todos os parâmetros disponíveis, a necessidade dos mesmos para o uso da ferramenta e uma breve descrição. Cada parâmetro esta coberto com mais detalhes a seguir.

O parâmetro #ArquivoFonte, é usado para informar o nome do arquivo a ser testado e deve ser sempre informado pois consiste de um parâmetro necessário.

Para os critérios implementados pela POKE-TOOL, o parâmetro #NomeFuncao deve ser configurado. Para a PROTEUM este parâmetro é desprezado.

O critério de teste é informado pelo parâmetro #Criterio e deve sempre estar presente. Os valores possíveis para este parâmetro são os nomes dos critérios: Todos-Nós, Todos-Arcos, Todos-Potenciais-Usos, Todos-Potenciais-Du-Caminhos, Todos-Potenciais-Usos/Du e Analise de Mutantes.

O parâmetro #CoberturaCriterio informa à ferramenta uma cobertura aceitável para o critério. Assim a execução pode ser encerrada ao alcançar tal cobertura. Se configurado com o valor -1, a ferramenta executará até que o máximo de gerações seja alcançado.

O tamanho da população é informado com o parâmetro #TamanhoPopulacao e deve ser portanto um valor maior que zero.

Para informar o número máximo de gerações é utilizado o parâmetro #NumeroGeracoes que deve ser um valor maior que zero. Este parâmetro pode ser configurado para zero, caso em que o processo de evolução não será realizado e apenas a população inicial será avaliada.

Tabela 5.3: Parâmetros detalhados

Parâmetro	Necessário	Detalhes
#ArquivoFonte	Sim	Nome do programa a ser testado.
#NomeFuncao	Sim	Nome da função a ser testada.
#Criterio	Sim	Critério de teste a ser utilizado.
#CoberturaCriterio	Sim	Valor α para a Cobertura desejável, $0 \geq \alpha \leq 1$.
#TamanhoPopulacao	Sim	Tamanho γ da população, onde $\gamma > 0$.
#NumeroGeracoes	Sim	Número máximo δ de gerações, $\delta \geq 0$.
#TaxaMutacao	Sim	Taxa ϵ de mutação para o AG, $0 \geq \epsilon \leq 1$.
#TaxaCrossover	Sim	Taxa ε de crossover para o AG, $0 \geq \varepsilon \leq 1$.
#Elitismo	Sim	Quantidade ζ de indivíduos que evoluem por elitismo, $0 \geq \zeta \leq \gamma$ e $\zeta + \eta + \theta = \gamma$
#Similaridade	Sim	Quantidade η de indivíduos que evoluem por similaridade, $0 \geq \eta \leq \gamma$ e $\eta + \zeta + \theta = \gamma$
#*Fitness*	Sim	Quantidade θ de indivíduos que evoluem por *fitness*, $0 \geq \theta \leq \gamma$ e $\theta + \zeta + \eta + = \gamma$
#FormatoEntrada	Sim	Tipos de dados da entrada do programa. I-int, F-float, D-double, C-char e S-*string*.
#NumeroArgumentos	Sim	Número β de parâmetros de entrada esperados como argumento, onde $\beta \geq 0$.
#ArquivoPopulacao	Não	Arquivo com dados semente para população inicial.
#AtivaTabu	Não	Ativar estratégia de lista Tabu.
#VariacaoInt	Não	Variação do tipo inteiro na forma $Lim_{inferior}/Lim_{superior}$. Default 0/10000.
#Variacao*String*	Não	Tamanho do tipo *string* na forma $Tam_{minimo}/Tam_{maximo}$. Default 1/30.
#TipoString	Não	Configura a geração de *string*.
#PausaGeracao	Não	Ativar pausa entre uma geração e outra.
#GeracoesComRepositorio	Não	Quantidade de gerações usando o repósitório de cobertura.

Para indicar a taxa de mutação o parâmetro #TaxaMutacao é utilizado. Conforme exposto na Tabela 5.3, deve conter um valor ϵ onde $0 \geq \epsilon \leq 1$.

Para indicar a taxa de crossover o parâmetro #TaxaCrossover é utilizado. Conforme também exposto na Tabela 5.3, deve conter um valor ε onde $0 \geq \varepsilon \leq 1$.

O parâmetro #Elitismo informa a quantidade de indivíduos da próxima geração gerados por elitismo.

O parâmetro #Similaridade informa a quantidade de indivíduos da próxima geração que devem ser gerados por similaridade.

#Fitness informa a quantidade de indivíduos da próxima geração gerados através do processo tradicional de evolução de AG's simples.

Estes três últimos parâmetros são complementares no sentido de que a soma de seus valores deve resultar no tamanho da população. Seus valores podem ser zero, desabilitando assim, a respectiva forma como os indivíduos da próxima geração são gerados.

O formato da entrada que o programa em teste espera deve ser informado através do parâmetro #FormatoEntrada. Seu valor deve ser os tipos de dados na ordem em que o programa os espera, sendo utilizada a convenção de I para *int*, F para *float*, D para *double*, C para *char* e S para *string*. Se o programa espera, por exemplo, um int, um char e um *string*, o conteúdo deste parâmetro será ICS.

O uso do parâmetro #NumeroArgumentos informa ao framework a quantidade n de entradas esperada pelo programa sendo testado em forma de argumento de chamada e não como entrada via teclado. Assim, os primeiros n tipos de dados configurados em #FormatoEntrada serão fornecidos para o programa em teste como argumento e o restante, se houver, será fornecido como entrada via teclado.

Todos os parâmetros citados até então são obrigatórios e necessários para o funcionamento da ferramenta.

O parâmetro #ArquivoPopulacao é opcional e possibilita o fornecimento de uma população inicial para o AG.

O parâmetro #AtivaTabu ativa o recurso de memorização disponível no framework. Este parâmetro é ativado com os valores "1" ou "sim" e desativado com qualquer outro

valor ou se não configurado.

O parâmetro #VariacaoInt é útil para limitar a variação do tipo inteiro para uma faixa de valores. Este recurso pode ser interessante para determinados tipos de programas. Seu valor deve estar no seguinte formato $Lim_{inferior}/Lim_{superior}$ e por default 0/10000 é assumido.

O parâmetro #VariacaoString limita a variação do tipo *string* para um tamanho mínimo e máximo. Seu valor deve estar no formato $Tam_{minimo}/Tam_{maximo}$ e por default 1/30 é assumido.

O parâmetro #TipoString customiza a geração aleatória para o tipo de dado *String*. Este parâmetro permite informar quais caracteres podem ser usados com a seguinte convensão: número (n), letras minúsculas (l), letras maiúsculas (L) e espaço em branco(s). Por default aceita tudo $slLn$.

O parâmetro #GeracoesComRepositorio é utilizado para balancear o ganho de desempenho com a utilização do repositório de execuções. Este repositório evita que um indivíduo já avaliado seja avaliado novamente caso esteja presente novamente em alguma geração. Como este recurso é implementado em arquivo, dependendo do número de gerações e do tamanho da população, a busca neste repositório pode se tornar mais lenta que executar o dado de teste novamente. Este parâmetro permite uma customização do uso deste recurso.

É possível que a ferramenta pare e pergunte ao testador se deseja continuar com a atividade de teste a cada geração. O parâmetro para esta funcionalidade é #PausaGeracao que deve conter o valor "1" para ser ativado.

O nome do arquivo com as configurações deve ser "arqconfig.txt" e deve estar no diretório onde se deseja utilizar a ferramenta. O arquivo fonte do programa em teste deve ser colocado em um subdiretório deste onde se encontra o arquivo de configuração, cujo nome deve ser "poketoolTest" por questões de compatibilidade com a Ferramenta POKE-TOOL.

Para facilitar a geração deste arquivo, esta disponível uma interface gráfica para elaboração do arquivo de configuração assim como para executar a Ferramenta *TDSGen*.

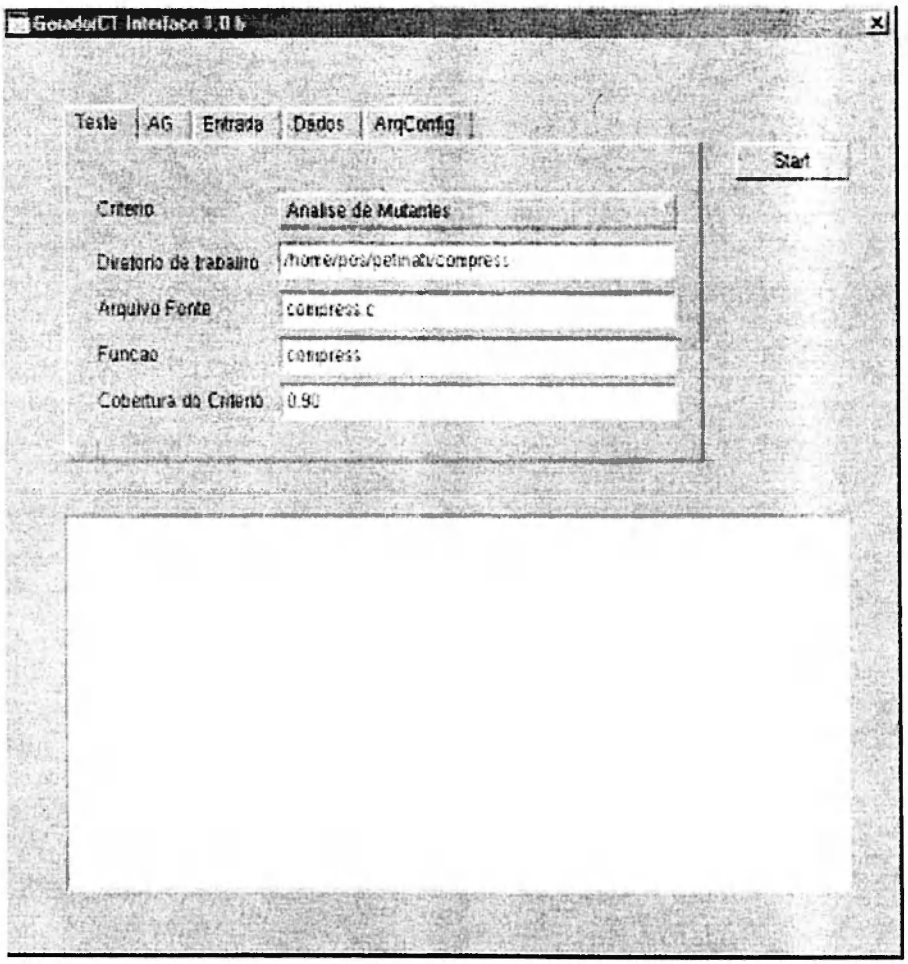

Figura 5.6: Interface gráfica da Ferramenta *TDSGen*

Para isso ela deve ser invocada a partir do diretório onde se deseja gerar o arquivo de configuração. Esta interface pode ser visualizada na Figura 5.6.

A interface é baseada em componentes de interface (abas) de fácil entendimento e navegação, possibilitando todas as configurações disponíveis e inclusive a visualização do arquivo de configuração gerado. Possui ainda, um componente de texto com barra de rolagens onde toda a saída do programa é apresentada.

5.3 Exemplo de Utilização

Esta seção descreve um exemplo prático de como se utiliza a ferramenta implementada.

O arquivo a ser testado neste exemplo foi compress.c. Este programa espera como

entrada um *string* (sequência de caracteres), informado via teclado e não por argumento, e fornece como saída uma representação compactada da entrada substituindo a repetição consecutiva de caracteres que ultrapassam três repetições por uma representação do tipo $\tilde{}\alpha\beta$, onde:

- α identifica o número de repetições através de letras do alfabeto atribuindo-se D para 4, E para 5 e assim sucessivamente até chegar em Z para 26 repetições. Ultrapassando este limite, o processo se repete caso as repetições da letra representada por β continuem e ultrapassem o mínimo de três repetições;
- β é o caractere que se repete.

Exemplos para o programa compress.c podem ser observados na Tabela 5.4.

Tabela 5.4: Saídas do programa compress.c

Entrada	Saída
abccccd	ab~Dcd
abcccccccd	ab~Gcd
abccccccccccccccccccccccccccccd	ab~Zc~Dcd

O critério de teste escolhido foi Análise de Mutantes da PROTEUM. Como mencionado acima, o programa compress.c necessita de um *string* como entrada, logo o formato da entrada para o programa é "S" e desta entrada, 0 (zero) é argumento. Por exclusão, a entrada do *string* deve ser feita via teclado e não argumento. A variação do tipo de dado *string* foi modificada para ter no mínimo 1 e no máximo 70 caracteres, as taxas de crossover e mutação foram configuradas para 90% e 10%, o tamanho da população a ser usado pelo AG é 50 indivíduos, o número de gerações foi definido para no máximo 100. Os indivíduos da próxima geração são gerados 10% por elitismo, 40% por *fitness* (AG tradicional) e 50% por similaridade. Para o exemplo, o recurso de lista Tabu está ativo e o número máximo de gerações que devem consultar o repositório de execuções é 90 das 100 gerações.

O arquivo de configuração para o exemplo com compress.c é apresentado na Figura 5.7. Elaborado o arquivo de configuração, basta executar a ferramenta com o comando

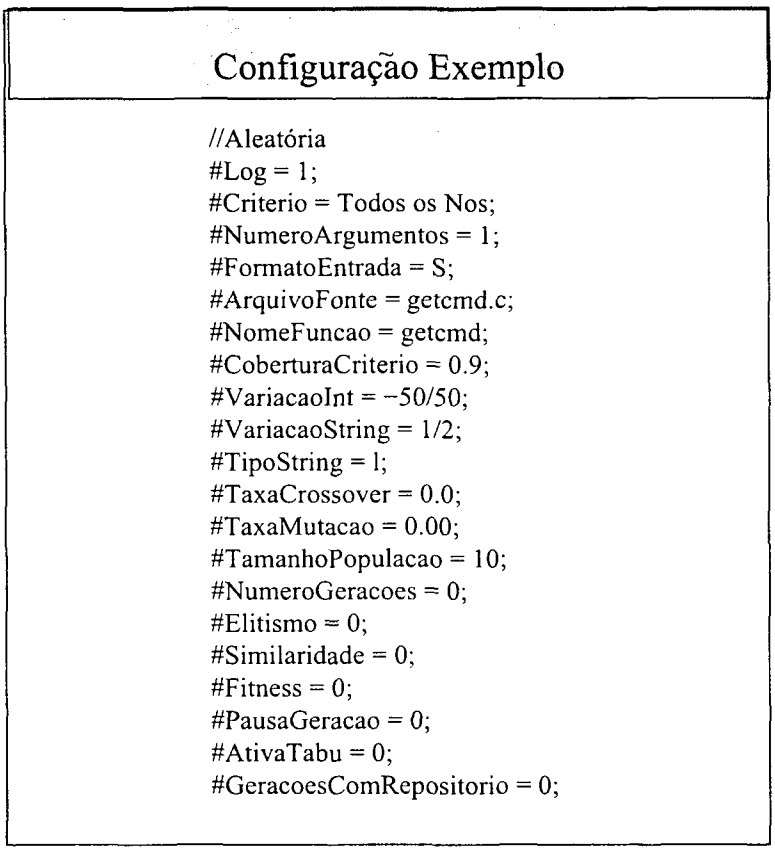

Figura 5.7: Ilustração de arquivo de configuração

"*tdsgen*". O resultado será um arquivo "Populacao.res" com 100 dados de teste gerado e outro "Tabu.res" com apenas os dados efetivos para a cobertura do critério. Para esta execução, a cobertura pode iniciar com aproximadamente 58% e chegar à 68%. Dados de execução e análise sobre os mesmos estão detalhados no Capítulo 6.

5.4 Considerações Finais

Nesse capítulo foi descrita a Ferramenta *TDSGen* que permite a integração com as ferramentas de teste POKE-TOOL e PROTEUM e a utilização dos critérios estruturais e baseado em erros repectivamente implementados por estas ferramentas.

A ferramenta implementa um AG e algumas estratégias tais como similaridade e Lista Tabu para aumentar o desempenho do algoritmo.

No próximo capítulo são descritos resultados de um experimento para validar a implementação da Ferramenta *TDSGen* e também das estratégias implementadas.

CAPÍTULO 6

EXPERIMENTO DE VALIDAÇÃO

O experimento descrito neste capítulo tem como objetivo validar a implementação da Ferramenta *TDSGen* e verificar a influência no desempenho do AG considerando as estratégias implementadas e descritas no Capítulo 5.

6.1 Descrição

Os programas utilizados no experimento e um breve descritivo de cada um estão na Tabela 6.1. Esses programas foram executados com os critérios implementados pelas Ferramentas de teste POKE-TOOL e PROTEUM, sendo eles: Todos-Nós, Todos-Arcos, Todos-Potenciais-Usos, Todos-Potenciais-Du-Caminhos, Todos-Potenciais-Usos/Du e Análise de Mutantes.

Tabela 6.1: Programas do experimento e descritivo

Programa	Descrição
compress.c	Obtém entradas do tipo *string* fornecidas via teclado e mostra na tela uma representação compactada substituindo caracteres repetidos mais de três vezes consecutivas por algo como $\tilde{}\alpha\beta$, onde α identifica o número de repetições através de letras do alfabeto e β identifica a letra.
getcmd.c	Este programa espera uma entrada do tipo *string* para comparar com um enumerator onde existem alguns comandos. Se o *string* fornecido corresponder a algum comando no enumerator, o programa devolve a posição correspondente ao comando no enumerator, caso contrário a posição para comandos desconhecidos será retornada.

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

- 10 execuções com geração de dados aleatória para três configurações diferentes;
- 10 execuções com AG simples para três configurações diferentes;

- 10 execuções com AG Híbrido para três configurações diferentes.

As execuções citadas acima foram realizadas utilizando-se a Ferramenta *TDSGen*, evitando assim, implementação extra para teste comparativo.

Para rodar a ferramenta gerando dados aleatoriamente, basta configurar o parâmetro #NumeroGeracoes com valor 0 (zero) não iniciando o processo de evolução com AG e não informar arquivo de população inicial com o parâmetro #ArquivoPopulacao fazendo com que a população inicial seja gerada aleatoriamente.

Para rodar a ferramenta evoluindo como um AG simples, basta não ativar o parâmetro #AtivaTabu, configurar #Elitismo e fazer #Similaridade 0 (zero).

Para rodar a ferramenta evoluindo como um AG híbrido, basta configurar os recursos ListaTabu, *fitness*, elitismo e similaridade respectivamente com os parâmetros #AtivaTabu, #Fitness, #Elitismo e #Similaridade.

A configuração utilizada para os experimentos é baseada nas configurações ilustradas pela Figura 6.1. Na figura, a Configuração A ilustra a configuração para geração aleatória para população de 10 indivíduos e critério Todos-Nós; a Configuração B mostra a configuração para geração com AG Simples de uma população de tamanho 50 e critério de teste Todos-Potenciais-Usos; a Configuração C ilustra a configuração para geração por AG Híbrido e popualação de 200 indivíduos com o critério Análise de Mutantes.

As Tabelas 6.2 e 6.3 apresentam as médias de cobertura alcançadas respectivamente para os programas getcmd.c e compress.c para as três estratégias propostas. Os dados destas tabelas estão representados graficamente nas Figura 6.2 e 6.3.

O tempo gasto (tempo de execução da ferramenta) com cada estratégia para cada programa está apresentado nas Tabelas 6.4 e 6.5 e, representados graficamente nas Figuras 6.4 e 6.5.

As Tabelas 6.6 e 6.7 apresentam os números de dados de teste efetivos para os dois programas; um dado efetivo é aquele que realmente contribui para o aumento da cobertura do critério.

Entretanto outros dados de teste foram gerados e consequentemente analisados pela ferramenta. Para se ter uma idéia da relação número de dados de teste analisados e

efetivos e de quanto a ferramenta pode ajudar o usuário nessa tarefa, esses números foram contabilizados para o programa getcmd.c. Esses números estão na Tabela 6.8.

Finalizando, a Tabela 6.9 mostra o resultado de execuções realizadas com 500 gerações para os critérios de teste Todos-Potenciais-Usos e Análise de Mutantes, a fim de se poder avaliar a influência desse parâmetro na cobertura atingida.

Configuração A	Configuração B	Configuração C
//Aleatória #Log = 1; #Criterio = Todos os Nos; #NumeroArgumentos = 1; #FormatoEntrada = S; #ArquivoFonte = getcmd.c; #NomeFuncao = getcmd; #CoberturaCriterio = 0.9; #VariacaoInt = −50/50; #VariacaoString = 1/2; #TipoString = l; #TaxaCrossover = 0.0; #TaxaMutacao = 0.00; #TamanhoPopulacao = 10; #NumeroGeracoes = 0; #Elitismo = 0; #Similaridade = 0; #Fitness = 0; #PausaGeracao = 0; #AtivaTabu = 0; #GeracoesComRepositorio = 0;	//AG Simples #Log = 1; #Criterio = Todos os Potenciais Usos; #NumeroArgumentos = 1; #FormatoEntrada = S; #ArquivoFonte = getcmd.c; #NomeFuncao = getcmd; #CoberturaCriterio = 0.9; #VariacaoInt = −50/50; #VariacaoString = 1/2; #TipoString = l; #TaxaCrossover = 0.9; #TaxaMutacao = 0.01; #TamanhoPopulacao = 50; #NumeroGeracoes = 100; #Elitismo = 0; #Similaridade = 0; #Fitness = 50; #PausaGeracao = 1; #AtivaTabu = 1; #GeracoesComRepositorio = 90;	//AG Híbrido #Log = 1; #Criterio = Todos os Potenciais Usos; #NumeroArgumentos = 1; #FormatoEntrada = S; #ArquivoFonte = getcmd.c; #NomeFuncao = getcmd; #CoberturaCriterio = 0.9; #VariacaoInt = −50/50; #VariacaoString = 1/2; #TipoString = l; #TaxaCrossover = 0.9; #TaxaMutacao = 0.01; #TamanhoPopulacao = 200; #NumeroGeracoes = 100; #Elitismo = 20; #Similaridade = 100; #Fitness = 80; #PausaGeracao = 1; #AtivaTabu = 1; #GeracoesComRepositorio = 90;

Figura 6.1: Configurações para experimentos

Tabela 6.2: Resultado para o programa getcmd.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	68.18%	68.63%	69.09%
	50	70.22%	69.31%	74.54%
	200	75.68%	75.00%	91.14%
Todos-Arcos	10	7.34%	9.33%	11.33%
	50	11.33%	10.67%	32.00%
	200	26.67%	36.67%	89.34%
Todos-Potenciais-Usos	10	7.34%	9.33%	9.33%
	50	8.02%	12.00%	30.67%
	200	28.00%	32.67%	90.00%
Todos-Potenciais-Du-Caminhos	10	7.34%	8.00%	9.34%
	50	14.00%	11.33%	40.00%
	200	31.33%	45.00%	90.66%
Todos-Potenciais-Usos/Du	10	8.67%	8.00%	11.33%
	50	13.34%	13.33%	40.00%
	200	32.00%	36.00%	88.67%
Análise de Mutantes	10	36.27%	34.44%	38.29%
	50	48.74%	48.75%	52.72%
	200	63.50%	63.10%	67.10%

Tabela 6.3: Resultado para o programa compress.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	93.75%	93.75%	93.75%
	50	93.75%	93.75%	93.75%
	200	93.75%	93.75%	93.75%
Todos-Arcos	10	83.33%	83.33%	83.33%
	50	83.33%	83.33%	83.33%
	200	83.33%	83.33%	83.33%
Todos-Potenciais-Usos	10	58.00%	58.00%	58.25%
	50	60.25%	59.75%	61.00%
	200	61.75%	60.50%	62.50%
Todos-Potenciais-Du-Caminhos	10	37.94%	37.06%	38.53%
	50	37.06%	42.94%	42.94%
	200	46.18%	46.00%	47.06%
Todos-Potenciais-Usos/Du	10	49.50%	49.00%	49.75%
	50	55.00%	56.50%	57.75%
	200	59.00%	57.50%	60.00%
Análise de Mutantes	10	42.77%	42.67%	42.82%
	50	42.91%	42.90%	43.02%
	200	43.16%	43.16%	43.27%

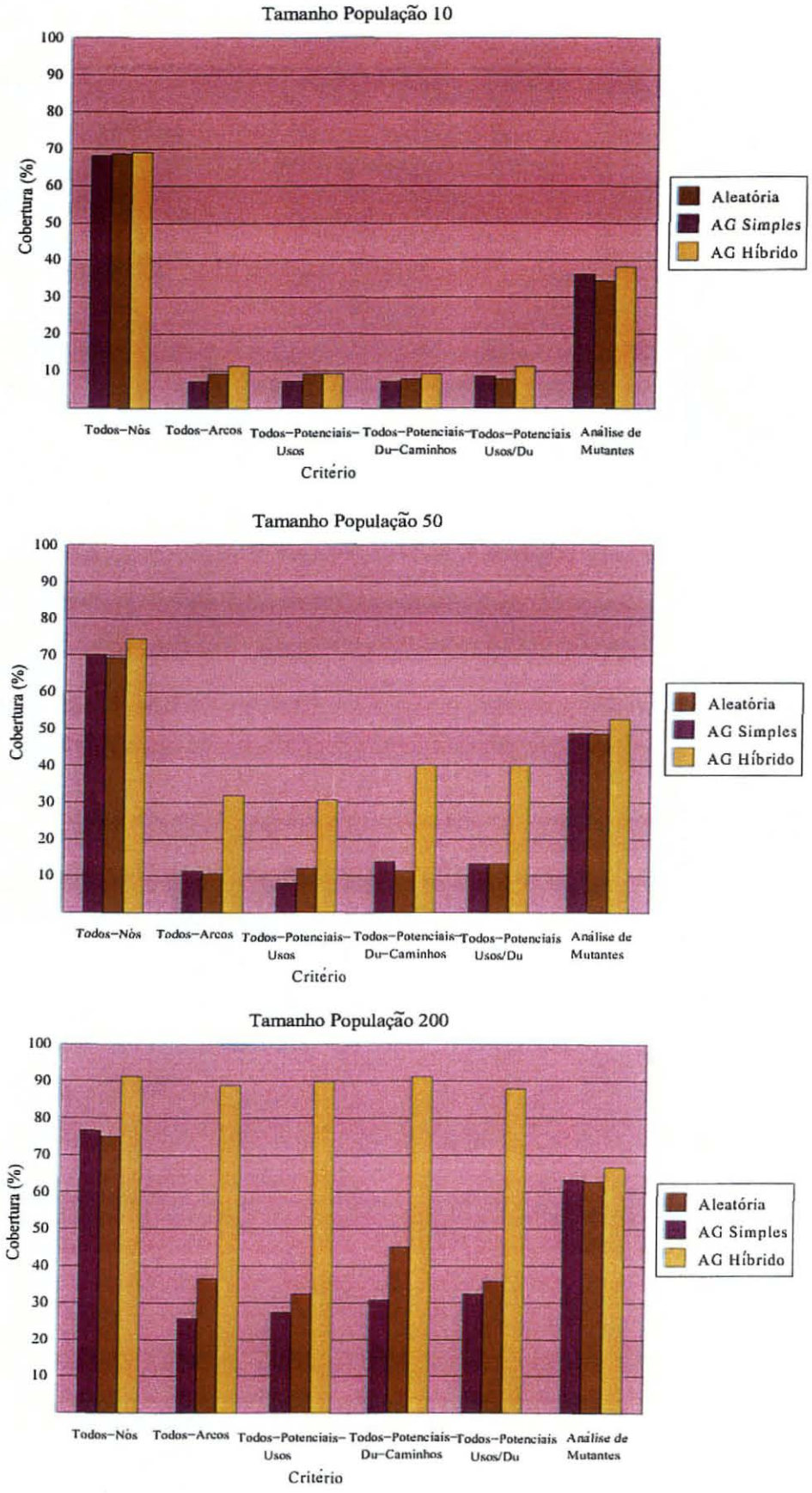

Figura 6.2: Gráficos de cobertura para Getcmd.c

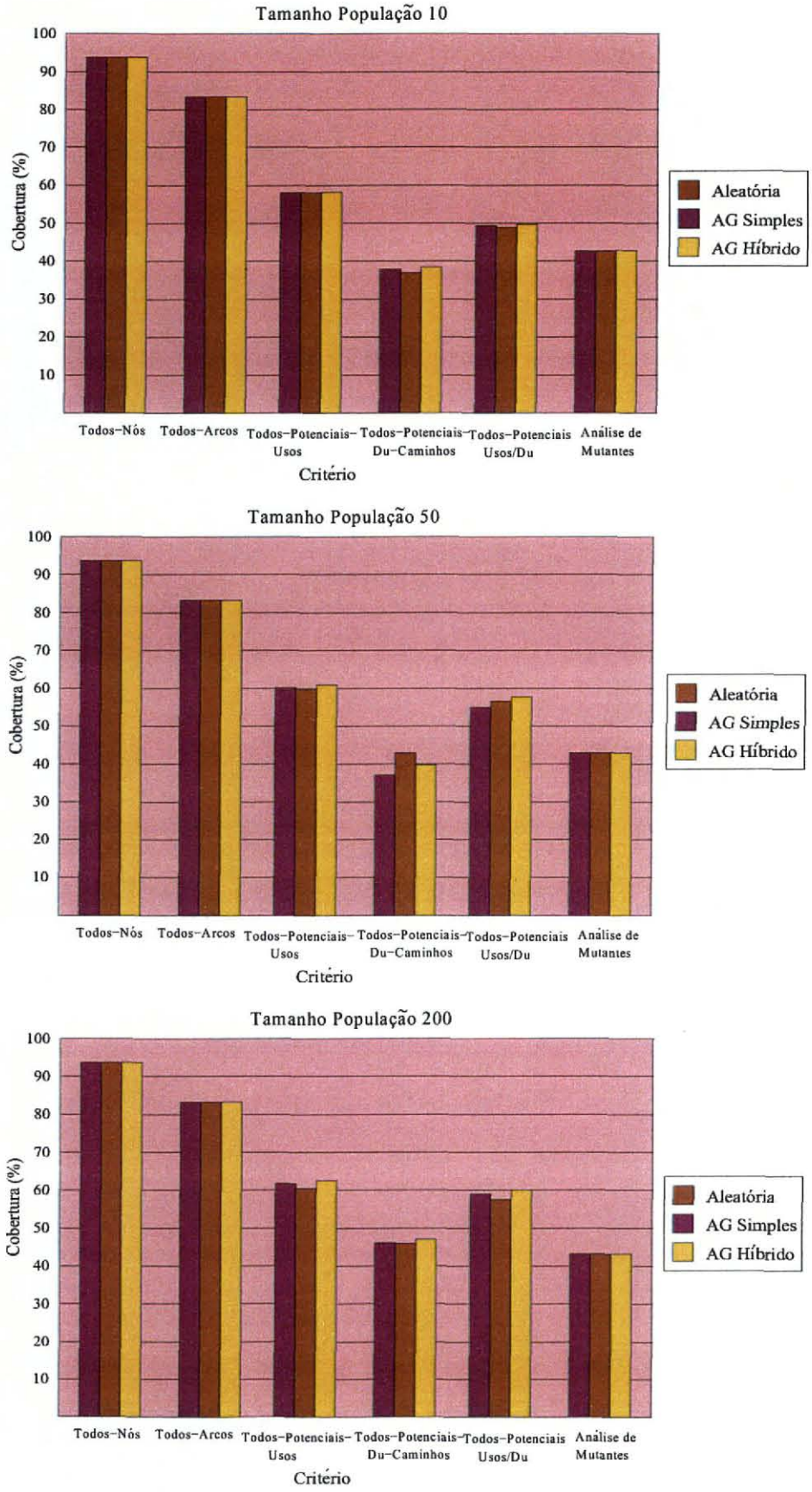

Figura 6.3: Gráficos de cobertura para Compress.c

Tabela 6.4: Tempo de execução para o programa getcmd.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	00:00:01	00:00:43	00:00:29
	50	00:00:04	00:02:02	00:01:49
	200	00:00:14	00:02:29	00:02:00
Todos-Arcos	10	00:00:01	00:04:46	00:01:33
	50	00:00:04	00:05:26	00:02:47
	200	00:00:13	00:09:21	00:10:51
Todos-Potenciais-Usos	10	00:00:01	00:03:54	00:01:26
	50	00:00:03	00:05:01	00:02:03
	200	00:00:13	00:07:25	00:06:47
Todos-Potenciais-Du-Caminhos	10	00:00:01	00:04:17	00:02:40
	50	00:00:07	00:04:48	00:05:08
	200	00:00:13	00:07:55	00:05:21
Todos-Potenciais-Usos/Du	10	00:00:01	00:04:05	00:01:36
	50	00:00:04	00:04:34	00:02:20
	200	00:00:14	00:07:52	00:06:59
Análise de Mutantes	10	00:02:56	00:26:10	00:12:34
	50	00:20:33	01:05:47	01:18:09
	200	01:23:06	02:09:43	02:43:53

Tabela 6.5: Tempo de execução para o programa compress.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	00:00:01	00:00:38	00:00:24
	50	00:00:03	00:03:07	00:01:55
	200	00:00:13	00:16:50	00:14:23
Todos-Arcos	10	00:00:01	00:00:45	00:00:24
	50	00:00:03	00:03:07	00:01:14
	200	00:00:14	00:16:33	00:14:33
Todos-Potenciais-Usos	10	00:00:01	00:00:52	00:00:28
	50	00:00:04	00:03:27	00:02:06
	200	00:00:14	00:18:09	00:13:56
Todos-Potenciais-Du-Caminhos	10	00:00:01	00:01:00	00:00:31
	50	00:00:04	00:03:17	00:01:46
	200	00:00:14	00:16:52	00:14:54
Todos-Potenciais-Usos/Du	10	00:00:01	00:00:56	00:00:30
	50	00:00:03	00:03:12	00:01:58
	200	00:00:15	00:17:41	00:15:38
Análise de Mutantes	10	00:05:50	00:54:25	00:47:25
	50	00:34:34	01:50:13	01:56:09
	200	02:14:42	04:01:04	04:18:14

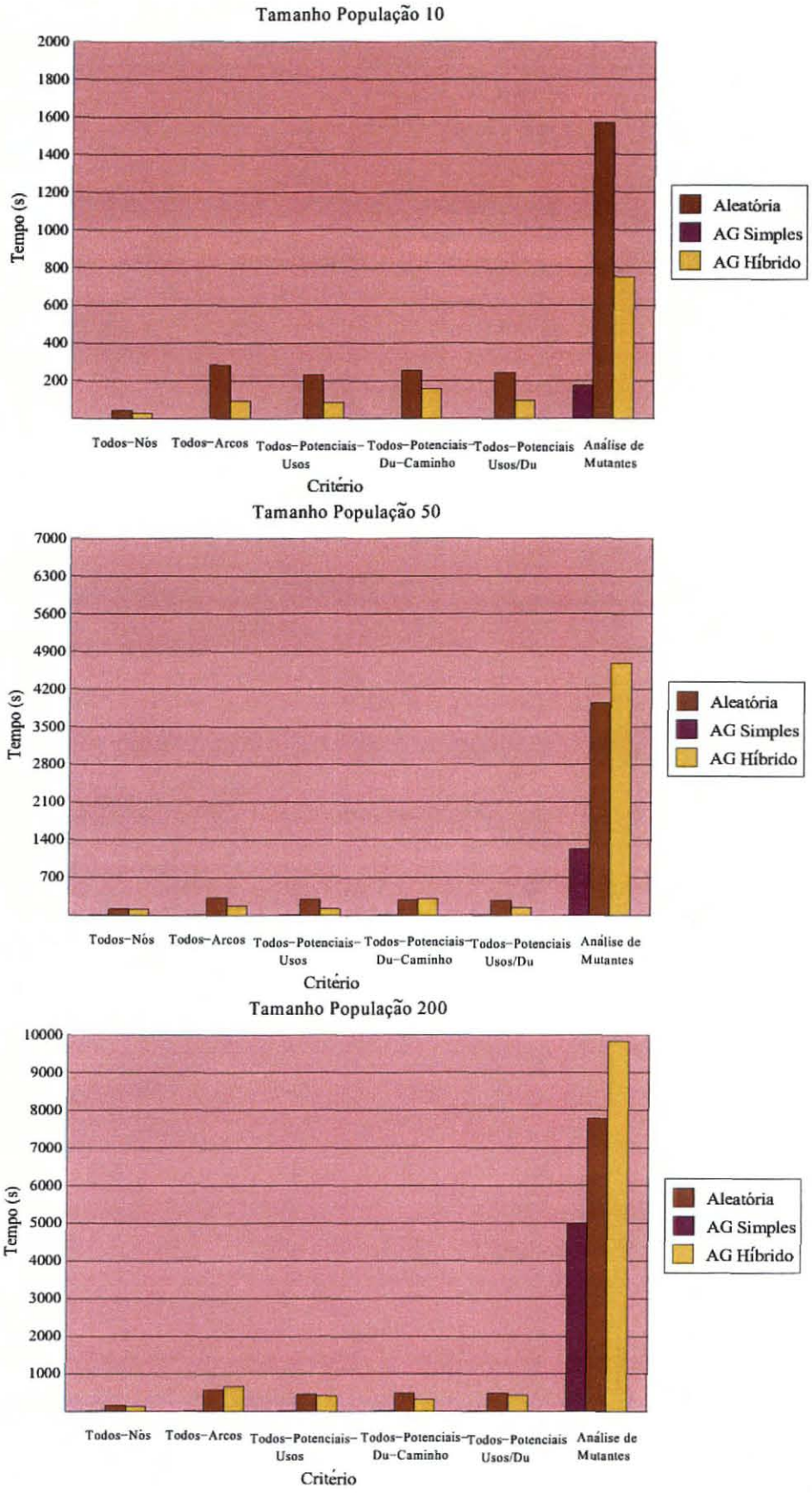

Figura 6.4: Gráficos de tempo para Getcmd.c

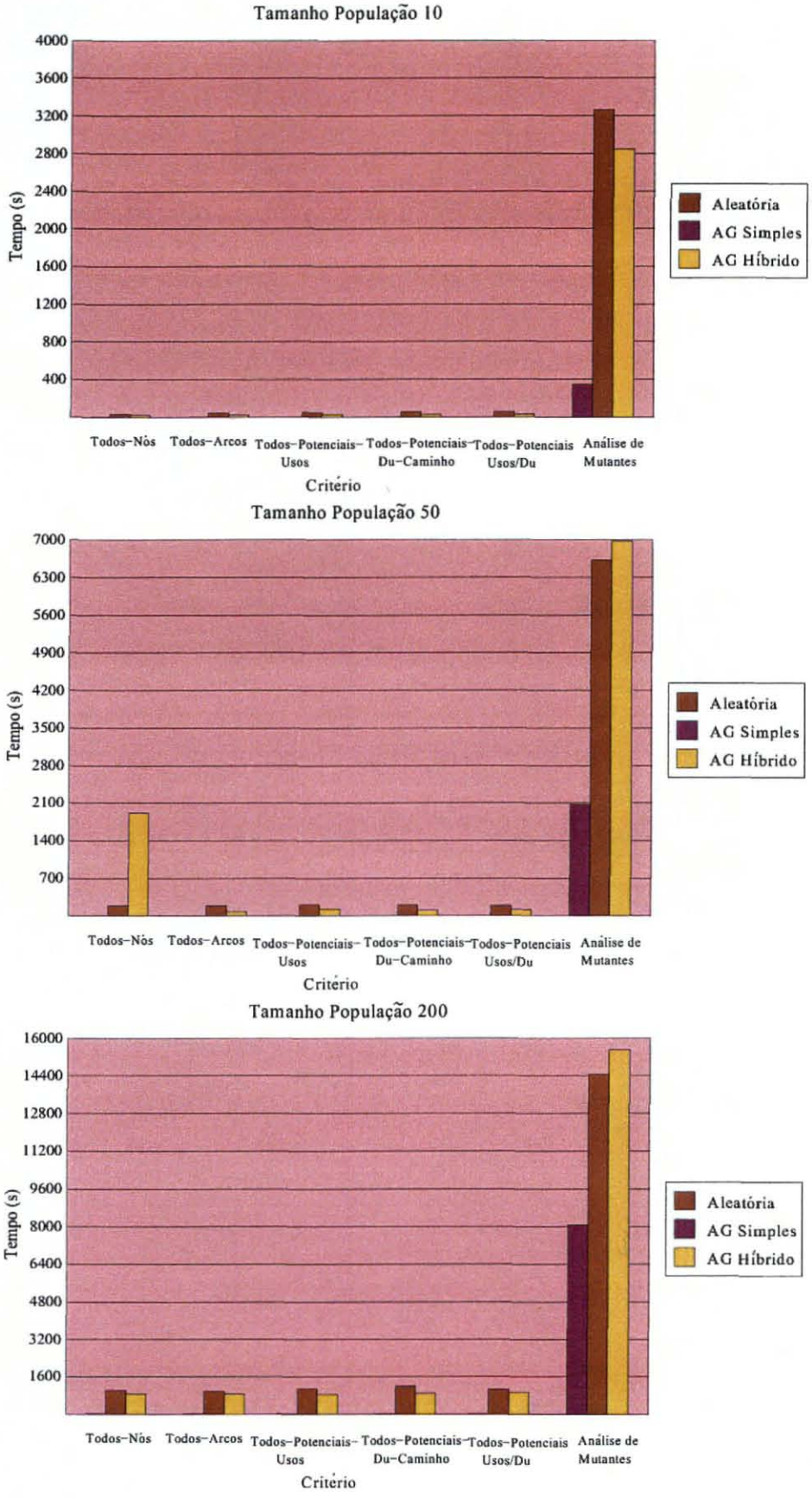

Figura 6.5: Gráficos de tempo para Compress.c

Tabela 6.6: Casos de teste para o programa getcmd.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	1	1	1
	50	3	3	4
	200	4	4	11
Todos-Arcos	10	1	2	2
	50	2	2	5
	200	4	6	13
Todos-Potenciais-Usos	10	1	1	1
	50	1	2	5
	200	4	5	14
Todos-Potenciais Du-Caminhos	10	1	1	1
	50	2	1	6
	200	5	7	14
Todos-Potenciais Usos/Du	10	2	2	2
	50	2	2	6
	200	6	8	13
Análise de Mutantes	10	10	8	11
	50	26	27	29
	200	37	33	40

Tabela 6.7: Casos de teste para o programa compress.c

Critério	Tam.População	Geração Aleatória	AG Simples	AG Híbrido
Todos-Nós	10	1	1	1
	50	1	1	1
	200	1	1	1
Todos-Arcos	10	1	1	1
	50	1	1	1
	200	1	1	1
Todos-Potenciais-Usos	10	1	1	1
	50	2	1	2
	200	2	2	3
Todos-Potenciais Du-Caminhos	10	2	2	2
	50	2	4	3
	200	3	3	4
Todos-Potenciais Usos/Du	10	2	2	2
	50	2	3	3
	200	4	3	4
Análise de Mutantes	10	2	2	2
	50	3	3	4
	200	6	6	7

Tabela 6.8: Comparativo entre o número de dados de teste gerados e efetivos para o programa getcmd.c com AG Híbrido

Critério	Tam.População	Gerados	Efetivos
Todos-Nós	10	230	1
	50	1233	4
	200	4788	11
Todos-Arcos	10	231	2
	50	1250	5
	200	4803	13
Todos-Potenciais-Usos	10	136	1
	50	753	5
	200	2807	14
Todos-Potenciais-Du-Caminhos	10	136	1
	50	742	6
	200	2805	14
Todos-Potenciais-Usos/Du	10	139	2
	50	740	6
	200	2418	13
Análise de Mutantes	10	146	11
	50	755	29
	200	2835	40

Tabela 6.9: Resultado para o programa getcmd.c com 500 gerações

Critério	Tam.População	Geração Aleatória	AG Híbrido
Potenciais-Usos	200	30.67%	93.33%
Análise de Mutantes	200	58.65%	68.95%

6.2 Análise dos Resultados

Com base nos dados apresentados na seção anterior, os seguintes aspectos foram levantados:

- Tempo: o tempo de evolução aumenta conforme o tamanho da população aumenta, como era de se esperar pois mais indivíduos terão que ser avaliados. O mesmo acontece se o número de gerações aumenta. A mudança do critério de teste, com excessão do critério Análise de Mutantes, não modifica o tempo de execução de forma significativa. Na comparação do tempo para a avaliação da primeira geração

com o tempo gasto para a avaliação para um número x de gerações, o tempo final não aumentou proporcionalmente ($tempo_{final} \neq x * (tempo_{inicial})$), fruto da estratégia de memorização implementada evitando a execução de um indivíduo que já foi avaliado em uma geração anterior. Na comparação dos tempos de AG simples e AG Híbrido, este último levou um tempo sensivelmente menor devido a estratégia de memorização. Esta mantém dados da cobertura de indivíduos em arquivo para obtê-las sem a necessidade de nova execução. Este recurso é muito útil para o critério Análise de Mutantes que possui o maior tempo para execução.

- Quantidade de dados de teste: com relação ao tamanho da população, para todos os critérios de teste analisados, quanto maior a população, maior o espaço de busca e maior o número de dados de testes efetivos. Quanto ao critério de teste utilizado, a quantidade de dados de teste não sofreu alterações significativas entre os critérios estruturais. Uma diferença é observada entre o critério Análise de Mutantes e qualquer outro critério. É importante ressaltar também que o número de dados de teste avaliados é sempre muito maior que a quantidade de dados efetivos resultantes. Este dado é apresentado na Tabela 6.8 e dá uma idéia de como a geração de dados de teste não é simples, podendo chegar a analisar em média 170 vezes o número de dados resultantes.

- Cobertura: para todos os critérios analisados, a cobertura aumenta conforme aumenta o tamanho da população. A cobertura encontrada para os critérios mais exigentes foi menor. O critério Todos-Potenciais-Du-Caminhos é o mais exigente deles. Essa exigência é dada por uma relação de inclusão entre critérios. Detalhes sobre essa relação de inclusão são apresentados em [22]. A Tabela 6.9 de execuções extras mostra resultados de um experimento realizado para o critério estrutural baseado em fluxo de dados Potenciais-Usos e o critério baseado em erros Análise de Mutantes com um número de 500 gerações. Esses resultados apontam que, com um número maior de gerações, a evolução da cobertura é mais significativa e promissora. A cobertura total do critério é difícil de ser conseguida devido a existência

de elementos requeridos não executáveis e mutantes equivalentes. Quanto à técnica utilizada para geração de dados de teste, a geração aleatória foi a que apresentou menor cobertura em média. Algumas vezes a técnica de AG simples apresentou um resultado levemente menor devido à probabilidade de perder bons indivíduos durante as gerações intermediárias no processo evolutivo - característica que garante que os AG's evitam ótimos locais. A técnica de AG Híbrido sempre apresentou o melhor resultado ou um resultado equivalente à alguma outra técnica. Este comportamento é devido ao mecanismo de memorização implementado para bonificar indivíduos com alto grau de similaridade, ou seja, indivíduos que influenciam fortemente na cobertura final da população perante o critério, e justifica a grande diferença de cobertura encontrada em algumas execuções.

6.3 Considerações Finais

Os resultados apresentados neste capítulo mostram que as estratégias implementadas pela Ferramente *TDSGen* aumentam o desempenho do AG Simples, sem entretanto aumentar o custo. Percebe-se inclusive uma redução no tempo de execução pois economiza-se tempo na avaliação da função de *fitness* de cada indivíduo.

Os resultados são muito interessantes e permitem alcançar uma cobertura satisfatória pois, em muitos casos, a existência de caminhos não executáveis não permite a completa satisfação dos critérios, ou seja, uma cobertura de 100%. Experimentos mostram que a maioria dos programas possuem caminhos não executáveis e a porcentagem de elementos requeridos é em torno de 10% [22], portanto uma cobertura de 80 a 90% está bem próxima do que se consegue satisfazendo os critérios completamente.

CAPÍTULO 7

CONCLUSÕES E TRABALHOS FUTUROS

Como já mencionado neste trabalho, a seleção de dados para atividade de teste pode ser uma tarefa demorada, complexa e custosa [44], o que pode acabar inviabilizando a aplicação de critérios de teste. Devido a isso, este trabalho assume um papel importante para tornar o teste de software uma atividade mais fácil.

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

Este trabalho contribui para o fortalecimento da pesquisa e para o estado da arte em teste de software, realizando estudos teóricos que validam abordagens e ferramentas já existentes além de desenvolver uma nova ferramenta de geração automática de dados de teste, utilizando e apresentando uma nova abordagem para tal finalidade voltados à satisfação de critérios de teste e não a uma meta de teste específica como um determinado caminho, comumente encontrado na literatura.

A Ferramenta *TDGen* implementa a nova abordagem proposta empregando a técnica de Algoritmos Genéticos com alterações e diferentes estratégias para melhor se ajustar ao problema de geração de dados de teste. Estas alterações, conhecidas como hibridização, incluem a implementação de lista Tabu, eliminação de duplicatas, bonificação por similaridade e memorização. A ferramenta proporciona também um ambiente com suporte à aplicação completa de uma estratégia de teste para critérios de técnicas diferentes, diferenciando-a de outras ferramentas disponíveis.

Outro diferencial é o fato de não exigir do testador uma análise do código em teste e como consequência aproximar a atividade de teste de pessoas sem profundos conhecimentos na área de teste de software.

Os resultados apresentados no Capítulo 6 demostram que AG é uma técnica promissora para a geração de dados de teste e que a utilização de outras estratégias com AG aumenta o

desempenho. Os AG's Híbridos foram os que obtiveram os melhores resultados em termos de cobertura e tempo de execução com relação à Geração Aleatória e AG Simples.

O mecanismo proposto de evolução com base em similaridade, bonificando e consequentemente mantendo indivíduos no processo de evolução do AG foi fundamental para o aumento da cobertura resultante.

7.1 Trabalhos Futuros

Como trabalhos futuros, podem ser realizadas diversas melhorias na Ferramenta *TDSGen* como a implementação de outros métodos de seleção e outras estratégias de hibridização, para a melhoria do desempenho. Outra tarefa a ser tratada é a identificação de elementos não executáveis. A implementação de heurísticas pode auxiliar a determinação dos mesmos e descontá-los da cobertura, assim, ficaria mais fácil avaliar os resultados e dados gerados pela ferramenta.

Poderiam ser implementados mecanismos para melhorar a interface da ferramenta com o usuário e novos experimentos com um conjunto maior de programas. Eles podem apontar novas direções de pesquisa para o teste evolucionário.

Elaborar novos experimentos para avaliar a Ferramenta *TDSGen* a eficácia, tempo e desempenho.

BIBLIOGRAFIA

[1] Hertz A, Taillard E, e Werra D. A tutorial on tabu search. Relatório técnico, Department of Matematich of Montreal University, 2000.

[2] Jones B, Eyres D.E, e Sthamer H.H. A strategy for using genetic algorithms to automte branch and fault-based testing. *The Computer Journal*, 41(2):98–107, outubro de 1998.

[3] Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering*, SE-16(8):870–879, agosto de 1990.

[4] Krishnamachari B. Global optimization in the design of mobile communication systems. Relatório técnico, Cornell University, 1999.

[5] Jones B.F, Sthamer H.H, e Eyres D.E. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 1996.

[6] Ombuki B.M., Nakamura M., e Osamu M. A hybrid search based on genetic algorithms and tabu search for vehicle routing. Relatório técnico, Department of Computer Science of Brock University, 2002.

[7] Michael C e et al. Genetic algorithms for dynamic test-data generation. *Technical Report*, 1997.

[8] Ramamoorthy C.V, Siu-Bun F.H, e Chen W.T. On automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, dezembro de 1976.

[9] Goldberg D.E e Richardson J. Genetic algorithms with sharing for multimodal function optimization. *Internationa Conference on Genetic Algorithms*, 1987.

[10] Glover F. Future paths for integer programming and links to artificial intelligence. *Computer and Operations Research*, 13:533–549, 1986.

[11] L.P Ferreira e S.R. Vergílio. A test data generation framework based on genetic algorithms. *4Th IEEE Latin-American Test Workshop*, 2003.

[12] Frankl F.G. *The Use of Data Flow Information for the Selection and Evaliation of Software Test Data.* Tese de Doutorado, Courant Institue of Mathematical Sciences, N.Y. University, 1987.

[13] McGraw G e Michael C. Automatic generation of test-cases for software testing. *Technical Report RST Corporation*, 1997.

[14] McGraw G, Michael C, e Schats M. Generating software test data by evolution. *Relatório Técnico*, 1998.

[15] McGraw G, Michael C, e Schats M. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, dezembro de 2001.

[16] Myers G. The art of software testing. *Wiley*, 1979.

[17] Agrawal H e et al. Mining systems tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, julho de 1998.

[18] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York, 1990.

[19] Chung I.S. Automatic testing generation for mutation testing using genetic. *Proceedings of SEKE*, junho de 1998.

[20] Goodenough J e Gerhart S.L. Toward a theroy of test data selection. *IEEE TSE*, 1975.

[21] Wegener J, Baresel A, e Sthamer H. Evolutionary test enviroment for automatic structural testing. *Information and Software Technology*, 2001.

[22] Maldonado J.C. Critérios potenciais usos: uma contribuição ao teste estrutural de software. *II International Conference of the SCCC*, 1991.

[23] Maldonado J.C., Chain M.L., e Jino M. Bridging the gap en the presence of infeaseble paths: Potential uses testing criteria. *II International Conference of the SCCC*, 1992.

[24] Baker J.E. Adaptive selection methods for genetic algorithms. *I International Conference on Genetic Algorithms and Their Applications*, 1985.

[25] Holland J.H. Adaptation in natural and artificial systems. *MIT Press*, 1992.

[26] Horgan J.R e London S. Data flow coverage and the c language. *IV Symp. Software Testing, Analysis, and Verification*, 1991.

[27] Duran J.W e Ntafos S.C. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, julho de 1984.

[28] Jong K.A. An analysis of the behavior of a class of genetic adaptive systems. *Tese de Doutorado*, 1975.

[29] Bottaci L. A genetic algorithm fitness function fo mutation testing. *IEEE International Conference on Software Engineering*, outubro de 2001.

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

[31] Davis L. Handbook of genetic algorithms. Relatório técnico, 1991.

[32] Mitchell M. Introduction to genetic algorithms. *MIT Press*, 1997.

[33] Roper M, Maclean I, Brooks A, e Miller J. Genetic algorithms and the automatic generation of test data. *Relatório Técnico*, 1995.

[34] Delamaro M.E e Maldonado J.C. Proteum - a tool for the assessment of test adequacy for c programs. *Manual do Usuário*, 1996.

[35] Chaim M.L. Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseados em análise de fluxo de dados. Relatório técnico, DCA/FEE/UNICAMP, 1991.

[36] Chaim M.L e Jino M. Manual de configuracao da poke-tool. Relatório técnico, DCA/FEE/UNICAMP, 1991.

[37] Tracey N, Clark J, Mander K, e McDermid J. An automated framework for structural test-data generation. *Proceedings of 13h IEEE Conference on Automated Software Engineering*, 1998.

[38] Bueno P.M.S. Geração automática de dados e tratamento de não executabilidade no teste estrutural de software. Dissertação de Mestrado, DCA/FEEC/Unicamp, 1999.

[39] Weichselbaum R. Software test automation by means of genetic algorithms. *Proceedings of the Sixth International Conference on Software Testing, Analysis and Review*, 1998.

[40] De Millo R.A, Gwind D.C, e King K.N. An extended overview of mothra software testing enviroment. *Second Workshop on Software Testing, Verification and Analysis*, páginas 142–151, julho de 1988.

[41] De Millo R.A, Lipton R.J, e Syward F.G. Hints on test data selection for the praticing programer. *Computer*, 1978.

[42] Pargas R.P, Harrold M.J, e Peck R.R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[43] Pressman R.S. Software engineering - a practitioner's approach. *McGraw-Hill*, 4, 1997.

[44] Pressman R.S. Software engineering. *Software-Pratice and Experience*, 2000.

[45] Rapps S e Weyuder E.J. Selection software test data using data flow information. *IEEE TSE*, 1985.

[46] Xanthakis S, Ellis C, Skourlas C, LeGall A, e Katsikas S. Application to genetic algorithms to software testing. *Proceedings of the Fifth International Conference on Software Engineering*, 1992.

[47] Chusho T. Test data selection test data selection and quality estimation based on the concept of essencial branches for path testing. *IEEE TSE*, 1976.

[48] McCabe T. A software complexity measure. *IEEE TSE*, 1976.

[49] Budd T.A. Mutation analisys: Ideas, examples, problems and prospects. *Computer Program Testing*, 1981.

[50] Miller W e Spooner D.L. Automatic generation of floating point test data. *IEEE TSE*, 1976.

[51] Howden W.E. Symbolic testing and dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, julho de 1977.

[52] Micheliwicz Z e Michaeliwicz M. Evolutionary computation techniques and their applications. *IEEE TSE*, 1997.

APÊNDICE A

FLUXO DE EXECUÇÃO DA *TDSGEN*

Este apêndice apresenta figuras que representam o fluxo de execução da Ferramenta *TDSGen*. A Figura A.1 apresenta a visão do processo como um todo. A Figura A.2 detalha a avaliação da população. A Figura A.3 ilustra a evolução de forma geral. O processo de evolução esta mais detalhado nas Figuras A.4, A.5 e A.6 tratando da evolução tradicional de AG, da evolução por elitismo e da evolução por similaridade respectivamente.

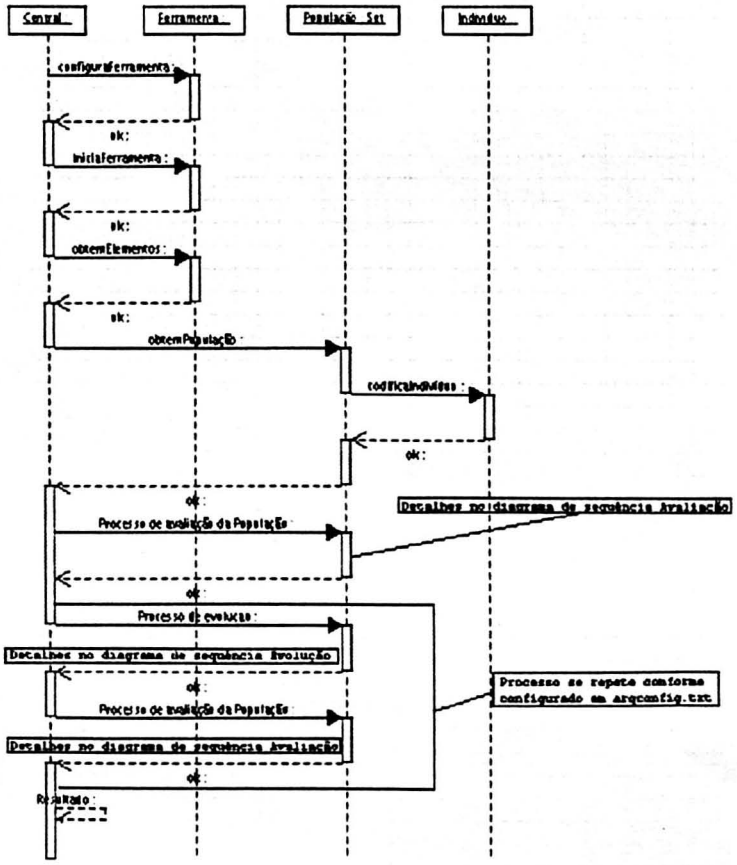

Figura A.1: Diagrama de seqüência da ferramenta - Global

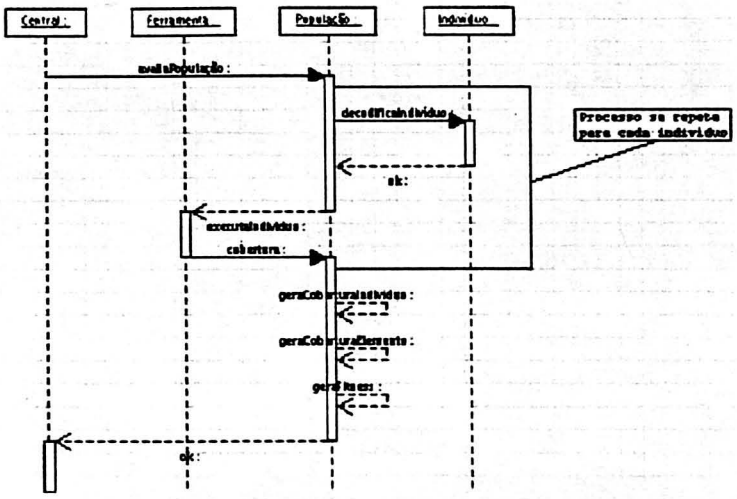

Figura A.2: Diagrama de seqüência da ferramenta - Avaliação

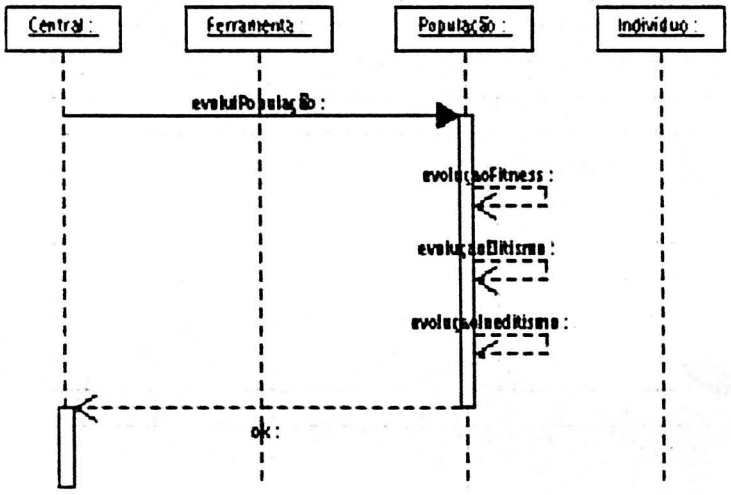

Figura A.3: Diagrama de seqüência da ferramenta - Evolução

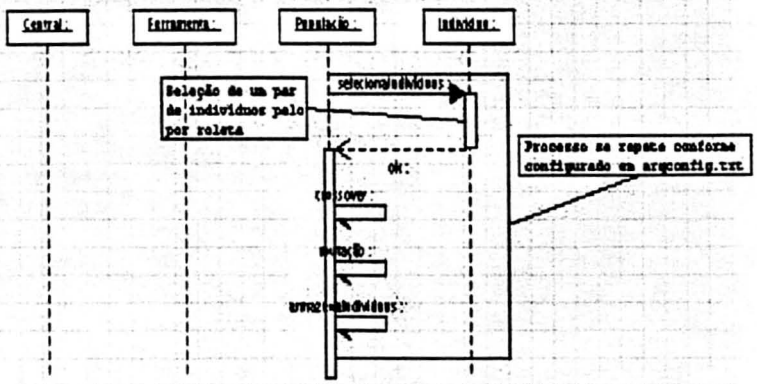

Figura A.4: Diagrama de seqüência da ferramenta - Evolução por *Fitness*

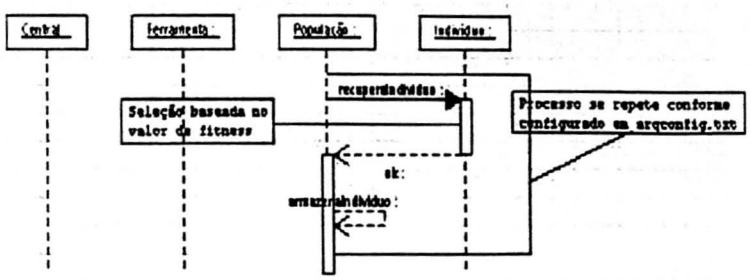

Figura A.5: Diagrama de seqüência da ferramenta - Evolução por Elitismo

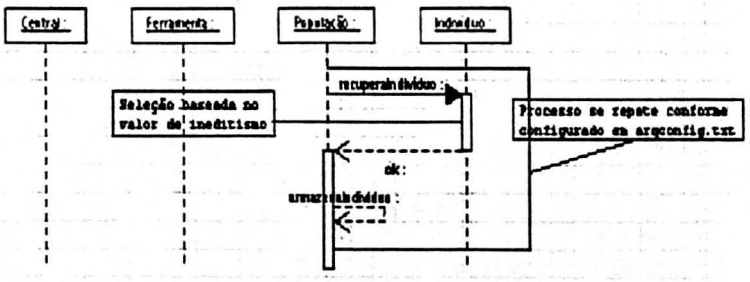

Figura A.6: Diagrama de seqüência da ferramenta - Evolução por Similaridade