UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Adilson de Almeida Neto

Desenvolvimento de uma Biblioteca para Geração Automática de Casos de Teste com Algoritmos Genéticos

DM: 14/23

Belém

2023

Adilson de Almeida Neto

Desenvolvimento de uma Biblioteca para Geração Automática de Casos de Teste com Algoritmos Genéticos

Dissertação de mestrado à Banca Examinadora do PPGEE da UFPA para obtenção do grau de Mestre em Engenharia Elétrica na Área de Computação Aplicada.

Universidade Federal do Pará

Orientador: Dr. Roberto Célio Limão de Oliveira

Coorientador: Dr. Rodrigo Lisbôa Pereira

Belém

2023

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD
Sistema de Bibliotecas da Universidade Federal do Pará
Gerada automaticamente pelo módulo Ficat, mediante os dados fornecidos pelo(a) autor(a)

N469d Neto, Adilson de Almeida.
Desenvolvimento de uma Biblioteca para Geração Automática de Casos de Teste com Algoritmos Genéticos / Adilson de Almeida Neto. — 2023.
65 f. : il. color.

Orientador(a): Prof. Dr. Roberto Célio Limão de Oliveira
Coorientador(a): Prof. Dr. Rodrigo Lisbôa Pereira
Dissertação (Mestrado) - Universidade Federal do Pará, , , Belém, 2023.

1. Algoritmo Genético. 2. Algoritmo de Busca. 3. Testes de Software. 4. Engenharia de Software. I. Título.

CDD 006.3

UNIVERSIDADE FEDERAL DO PARÁ
INSTITUTO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"DESENVOLVIMENTO DE UMA BIBLIOTECA PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE COM ALGORITMOS GENÉTICOS"

AUTOR: **ADILSON DE ALMEIDA NETO**

DISSERTAÇÃO DE MESTRADO SUBMETIDA À BANCA EXAMINADORA APROVADA PELO COLEGIADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA, SENDO JULGADA ADEQUADA PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA NA ÁREA DE COMPUTAÇÃO APLICADA.

APROVADA EM: 20/04/2023

BANCA EXAMINADORA:

____

Prof. Dr. Roberto Célio Limão de Oliveira
(Orientador – PPGEE/UFPA)

____

Prof.ª Dr.ª Jasmine Priscyla Leite de Araújo
(Avaliadora Interna – PPGEE/UFPA)

____

Prof. Dr. Otávio Noura Teixeira
(Avaliador Externo ao Programa – CAMPUS TUCURUÍ/UFPA)

____

Prof. Dr. Rodrigo Lisbôa Pereira
(Avaliador Externo – UFRA)

VISTO:

____

Prof. Dr. Diego Lisboa Cardoso

(Coordenador do PPGEE/ITEC/UFPA)

Resumo

Neste trabalho uma biblioteca é desenvolvida com o proposito de gerar casos de teste automaticamente na linguagem de programação Python, para a geração dos testes, é utilizado um algoritmo genético com um operador de mutação desenvolvido de forma ad-hoc baseado na interação social. O Algoritmo é aplicado ao problema de geração de dados para testes com sucesso, resultados razoáveis são obtidos quando comparado ao estado da arte, o que demonstra um possível caminho a ser explorado na solução deste tipo de problema.

Palavras-chave: Algoritmo Genético. Algoritmo de Busca. Testes de Software. Engenharia de Software.

Abstract

In this work a library capable of automatically generating test cases for the Python programming language is developed, these tests are generated utilizing an genetic algorithm which uses an ad-hoc mutation operator based on social interaction. The algorithm is applied with success to the problem of automatically generating software tests, showing promising results when compared to the state of the art, revealing itself as a possible path of exploration when solving this category of problem.

Keywords: Genetic Algorithm. Search Algorithm. Software Testing. Software Engineering.

Lista de ilustrações

Figura 1. Fluxograma ilustrando o algoritmo genético 15
Figura 2. Exemplificação do Cruzamento entre dois indivíduos 17
Figura 3. Ilustração do Teste Evolutivo retirado de (SCHLINGLOFF; VOS; WEGENER, 2023) 23
Figura 4. Visualização da função *Fitness* _linearize 40
Figura 5. Visualização da função *Fitness* rgb_to_hsv - Nó 1 41
Figura 6. Visualização da função *Fitness* rgb_to_hsv - Nó 2 41
Figura 7. Visualização da função *Fitness* rgb_to_hsv - Nó 3 42
Figura 8. Visualização da função *Fitness* rgb_to_hsv - Nó 4 42
Figura 9. Diversidade por Algoritmo 47
Figura 10. Cobertura Média Alcançada por Algoritmo 47
Figura 11. Média de Gerações Até o Ótimo 48
Figura 12. Tempo Despendido por Algoritmo por Problema 48

Lista de tabelas

Tabela 1. Tabela de Pagamentos do Jogo Dilema do Prisioneiro 26
Tabela 2. Repositórios Usados na Base de Dados 39
Tabela 3. Base de Dados de Funções 39
Tabela 4. Resultados Tabelados 46

Lista de abreviaturas e siglas

AG	Algoritmo Genético
SBT	*Search Based Testing*
GASI	*Genetic Algorithm with Social Interaction*
PSO	*Particle Swarm Optimization*
SLA	*Service Level Agreement*
PD	*Prisoner Dilemma*
MMIS	Mutação Moderada por Interação Social

Sumário

1 INTRODUÇÃO . . . 12
1.1 Objetivo Geral . . . 13
1.2 Objetivos Específicos . . . 13
1.3 Organização do Trabalho . . . 13

2 EMBASAMENTO TEÓRICO . . . 14
2.1 Algoritmos Genéticos . . . 14
2.1.1 Composição do Indivíduo . . . 14
2.1.2 Operador de Seleção . . . 14
2.1.3 Operador de Cruzamento . . . 16
2.1.4 Operador de Mutação . . . 17
2.1.5 Elitismo . . . 17
2.1.6 Fluxo Geral . . . 18
2.1.7 Subpopulações e Migrações . . . 18
2.1.8 Diversidade . . . 18
2.1.8.1 Distância de Hamming . . . 18
2.1.8.2 Distância Euclidiana . . . 19
2.1.8.3 Distância de Manhattan . . . 19
2.2 Busca Aleatória . . . 19
2.3 Testes de Software . . . 19
2.3.1 Testes Caixa Branca . . . 20
2.3.2 Testes Caixa Preta . . . 21
2.4 Testes Automatizados . . . 21
2.4.1 Testes de Unidade . . . 21
2.4.2 Teste de Integração . . . 21
2.4.3 Teste de Sistema . . . 22
2.4.4 Teste de Aceitação . . . 22
2.5 Teste Baseado em Busca . . . 22
2.6 Teste Evolutivo . . . 23
2.7 Teoria dos Jogos . . . 24
2.7.1 Estratégia e Estratégia Dominante . . . 25
2.7.2 Dilema do Prisioneiro . . . 25
2.7.3 Pagamento ou *Payoff* . . . 25
2.8 Estado da Arte - Algoritmo Genético DaimlerChrysler . . . 26
2.8.1 Mutação no DaimlerChrysler . . . 26

2.8.2 Seleção no DaimlerChrysler . . . 26
2.8.3 Cruzamento no DaimlerChrysler . . . 27
2.8.4 Migrações no DaimlerChrysler . . . 27
2.9 Trabalhos Relacionados . . . 27
2.10 Resumo do Capítulo . . . 29

3 BIBLIOTECA PARA GERAR CASOS DE TESTE AUTOMATICAMENTE COM ALGORITMOS GENÉTICOS . . . 30
3.0.1 Funções Suportadas . . . 30
3.0.2 Instrumentação das Funções . . . 31
3.0.3 Otimizadores . . . 33
3.0.4 Processamento de Resultados . . . 33
3.1 Mutação Baseada em Interação Social . . . 33
3.1.1 Fase de Interação Social . . . 34
3.1.2 Utilização do Valor de *Payoff* na Mutação . . . 34
3.2 Operadores do Algoritmo Genético . . . 35
3.2.1 Função *Fitness* . . . 35
3.3 Taxonomia do Indivíduo . . . 37
3.4 Escolha de Parâmetros . . . 37
3.5 Resumo do Capítulo . . . 37

4 ESTUDO DE CASO . . . 38
4.1 Geração da Base de Dados . . . 38
4.2 Instrumentação da Base de Dados . . . 39
4.3 Visualização do *Fitness* . . . 40
4.3.1 Função _linearize . . . 40
4.3.2 Função rgb_to_hsv . . . 40
4.4 Metrificação da Diversidade . . . 43
4.5 Justificativa . . . 43
4.6 Limitações . . . 44
4.7 Resumo do Capítulo . . . 44

5 RESULTADOS OBTIDOS . . . 45
5.1 Ambiente de Programação e Detalhes de Hardware . . . 45
5.2 Algoritmos Comparados . . . 45
5.3 Cobertura Encontrada . . . 45
5.4 Diversidade . . . 46
5.5 Velocidade de Convergência . . . 47
5.6 Resumo do Capítulo . . . 49

6 CONSIDERAÇÕES FINAIS . . . 50

6.1 Trabalhos Anteriores **50**
6.2 Dificuldades Encontradas **51**
6.2.1 Instrumentação das Funções 51
6.2.2 Construção da Base de Dados 51
6.2.3 Tempo de Execução do Algoritmo 51
6.3 Trabalhos Futuros **52**

REFERÊNCIAS **53**

A ARTIGO ERIM **56**

B PAPER LA-CCI **60**

1 Introdução

Testes são uma parte essencial do ciclo de desenvolvimento de software, uma vez que a falta de testes é uma causa provável de falhas de software (HARMAN; MCMINN, 2009) e por este motivo mais investimentos são necessários para obter testes mais robustos.

Estima-se que 30% a 50% do esforço total no desenvolvimento de software é dedicado para a criação de testes (ELLIMS; BRIDGES; INCE, 2006), o que é razoável, dado que falhas de software causam problemas para os clientes e desenvolvedores e a depender do tipo de software, podem causar danos mais graves.

Para uma equipe de desenvolvimento de software, cada hora dedicada a testes é uma hora que não é dedicada ao desenvolvimento do produto, a fonte real da receita de uma companhia. Devido a este custo de produtividade, novos métodos são desejáveis e a possibilidade de testes gerados de forma automática ou semi-automática são valiosos (MCMINN, 2011). Ferramentas que podem auxiliar a geração de testes podem ter uma excelente contribuição na produtividade dos desenvolvedores, uma vez que se é possível gerar testes com a mesma qualidade com um tempo menor, mais horas serão dedicadas ao desenvolvimento.

A criação de uma ferramenta desse tipo não é simples, uma vez que uma solução extremamente focada pode não ser útil numa gama grande de problemas, enquanto uma solução mais genérica pode perder efetividade e gerar ganhos menores de produtividade. A flexibilidade e aplicabilidade dos Algoritmos Genéticos (AG) fazem desta meta-heurística uma boa candidata para soluções boas o suficiente para uma gama maior de problemas.

Porém, antes de se determinar que tipo de solução aplicar, primeiro é importante definir que tipo de testes serão gerados por uma ferramenta qualquer, o problema central tratado por este trabalho é a geração de dados utilizados em testes.

A aplicabilidade do AG na geração de dados de teste é bem estudada(HARMAN; MCMINN, 2009) e neste trabalho, é proposto o desenvolvimento de um operador de mutação, chamado de aqui de Mutação Moderada por Interação Social (MMIS), que poderá ser utilizado nesta geração de dados.

Além disso, também é proposto o desenvolvimento de uma biblioteca na linguagem Python, para a geração de casos de teste que utiliza este e outros algoritmos.

O operador de mutação proposto é baseado no Genetic Algorithm with Social Interaction (GASI) (PEREIRA et al., 2020) e apesar da inspiração se tratam de algoritmos diferentes, o GASI é uma versão modificada do Algoritmo Genético, enquanto este trabalho apresenta apenas um operador de mutação diferente que pode ser aplicado em um AG

comum.

Especificamente, esse operador MMIS é aplicado ao problema de geração de dados de teste comparado ao estado da arte, o DaimlerChrysler (HARMAN; MCMINN, 2009), chegando a resultados promissores.

1.1 Objetivo Geral

Propor uma ferramenta para geração automática de dados de testes, contendo um operador de mutação baseado no Algoritmo Genético com Interação Social (GASI) (PEREIRA et al., 2020) e avaliando o desempenho comparado ao algoritmo do estado da arte.

1.2 Objetivos Específicos

- Criar uma base de dados com funções de teste a serem utilizadas no trabalho.
- Desenvolver a biblioteca de geração de casos de teste a ser utilizado neste trabalho.
- Desenvolver um Algoritmo Genético com operador de mutação baseado em Interação Social na linguagem de programação Python.
- Realizar o estudo e comparar o operador de mutação desenvolvido neste trabalho ao estado da arte para uma seleção de funções de teste instrumentadas.

1.3 Organização do Trabalho

O trabalho aqui exposto é dividido em capítulos, e sua organização é feita da forma a seguir:

- No capítulo 2 é apresentado o embasamento teórico do trabalho, que será a base dos capítulos seguintes.
- No capítulo 3 a ferramenta desenvolvida é apresentada em conjunto com sua implementação.
- No capítulo 4, um estudo de caso é demonstrado com fim de validar a ferramenta e o algoritmo proposto neste trabalho.
- No capítulo 5, os resultados obtidos são avaliados.
- Por fim, no capítulo 6, são feitas as considerações finais acerca do trabalho.

2 Embasamento Teórico

2.1 Algoritmos Genéticos

O algoritmo genético (AG) foi desenvolvido em 1975 por John Holland (HOLLAND, 1992) na Universidade de Michigan e a técnica consiste em usar o conceito de sobrevivência do mais apto da Teoria Evolutiva Darwiniana em um algoritmo de busca, onde várias soluções em potencial competem pelo direito de existir na população, onde soluções mais "aptas"ou com maior *Fitness* tem maior chance de continuar a existir.

Na figura 1, é possível observar o fluxograma padrão do algoritmo genético, onde estão ilustrados três operadores: Seleção, Cruzamento e Mutação.

2.1.1 Composição do Indivíduo

O algoritmo genético é composto por uma série de indivíduos que são potenciais soluções para um dado problema, avaliados por uma função *Fitness*. Os valores associados a um indivíduo são chamados de cromossomo (BEASLEY; BULL; MARTIN, 1993), no caso deste trabalho, estão sendo tratados problemas de variáveis reais, ou seja, problemas nos quais cada solução consiste em uma sequência de números reais.

Assim sendo, a população pode ser vista como uma grande matriz, com dimensões M por N, onde M é a dimensionalidade do problema, ou seja, a quantidade de números reais que uma solução requer e N é a quantidade de indivíduos.

2.1.2 Operador de Seleção

O processo de seleção é um passo importante do algoritmo genético (MITCHELL, 1998).

Um dos métodos de seleção utilizados em algoritmos genéticos é chamado de seleção proporcional ao *fitness*, também conhecido como seleção por roleta. Esse método consiste em assimilar uma probabilidade de seleção para cada indivíduo da população baseado no valor *fitness*, o quão bem a solução resolve o problema em questão. Indivíduos com um valor de *fitness* elevados tem chances maiores de serem selecionados, enquanto os que tem menor valor tem uma probabilidade menor de serem selecionados (SHUKLA; PANDEY; MEHROTRA, 2015).

Para implementar a seleção por roleta, os valores de *fitness* individuais são primeiramente normalizados de forma que a soma de todos esses valores seja um, isso é feito ao se dividir cada valor pela soma de todos os valores na população. Em seguida, uma "roleta"é

Figura 1. Fluxograma ilustrando o algoritmo genético

construída dividindo várias cunhas ou pedaços da roleta onde cada cunha corresponde a um indivíduo da população. O tamanho dessas cunhas é proporcional ao valor normalizado do *fitness* que representa cada indivíduo.

Para selecionar o individuo um valor aleatório é gerado entre 0 e 1, e o individuo correspondente a cunha é escolhido. Esse processo se repete quantas vezes forem necessárias (SHUKLA; PANDEY; MEHROTRA, 2015).

Outro método comum de seleção é chamado de seleção baseada em ranqueamento. Neste método, os indivíduos da população são ranqueados ou ordenados de acordo com seu valor *fitness*, onde o maior valor tem o primeiro ranque e o menor tem o último ranque. Probabilidades então são associadas a cada ranque de forma que os primeiros ranques tem maior probabilidade de seleção (SHUKLA; PANDEY; MEHROTRA, 2015).

Note que a principal diferença entre o ranque e a roleta é que a roleta faz a seleção ser diretamente proporcional ao *fitness*, então se o desempenho de um indivíduo for igual a soma de todos os outros, este tem 50% de chance de ser escolhido, enquanto no sistema

de ranqueamento, ele simplesmente tem o percentual associado ao primeiro ranque. Dessa forma, indivíduos com um percentual maior de *fitness* quando comparados ao resto da população tem uma vantagem na seleção por ranqueamento.

Existe um terceiro método de seleção para algoritmos genéticos relevante para este trabalho, a seleção por torneio. Nesse processo de seleção, vários indivíduos são retirados aleatoriamente da população e o mais apto entre todos eles é selecionado. O tamanho do torneio, aqui denominado J tem um papel importante neste processo de seleção (SHUKLA; PANDEY; MEHROTRA, 2015).

Quando o tamanho do torneio se aproxima do tamanho da população, mais este se aproxima de favorecer indivíduos mais aptos, como um exemplo, um torneio do tamanho da população sempre seleciona o individuo mais apto, por outro lado, um torneio pequeno demais é excessivamente aleatório. Desta forma, é possível ajustar o tamanho J do torneio para alcançar uma seleção que é mais elitista ou busca mais a diversidade, assim como a seleção por ranqueamento, este algoritmo confere flexibilidade.

2.1.3 Operador de Cruzamento

O processo de cruzamento, também chamado de *crossover* ou recombinação, é um processo utilizado em um algoritmo genético no qual uma nova população ou uma *offspring* é gerada a partir de soluções existentes numa população.

É um passo importante na otimização do AG, já que é responsável por combinar a informação genética dos pais com o objetivo de gerar indivíduos potencialmente melhores (UMBARKAR; SHETH, 2015).

Existem várias técnicas de *crossover* que podem ser usados em AGs, dentre as quais pode se citar o cruzamento de um ponto. Nesta técnica, um único ponto de corte é escolhido separando os cromossomos dos dois pais em quatro pedaços diferentes, que serão recombinados para formar dois indivíduos filhos (UMBARKAR; SHETH, 2015). O processo pode ser observado na Figura 2.

Outra forma de cruzamento é a denominada cruzamento uniforme. Nesta técnica, cada filho é gerado ao se iterar por todos os seus cromossomos e preenchendo o valor a partir de um pai ou de outro com probabilidade igual (UMBARKAR; SHETH, 2015).

O cruzamento é aplicado com uma probabilidade P_c (MITCHELL, 1998). Um valor mais alto de P_c significa que o cruzamento é mais constante, enquanto um valor de cruzamento menor diminui a quantidade de cruzamentos. O valor de P_c pode ser determinado por meio da experimentação ou pelas características do problema, resolvido pelo algoritmo genético.

Por fim, o objetivo do cruzamento é combinar as soluções selecionadas com o

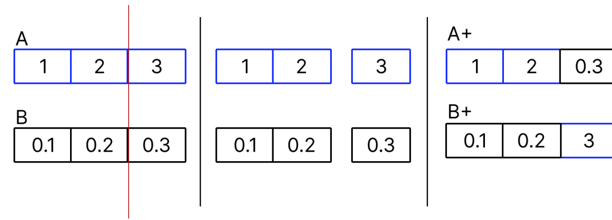

Figura 2. Exemplificação do Cruzamento entre dois indivíduos

objetivo de gerar filhos melhores a partir dos indivíduos existentes na população, portanto, o cruzamento tem um carácter combinatório.

2.1.4 Operador de Mutação

A mutação é um processo no qual o algoritmo genético introduz diversidade genética numa população de soluções ou indivíduos, ao fazer mudanças aleatórias em seus cromossomos (SRINIVAS; PATNAIK, 1994). É um passo importante no processo de otimização do AG, uma vez que ele ajuda a explorar novos locais do espaço de busca que pode levar a descoberta de soluções melhores do que as já encontradas até o momento (MITCHELL, 1998).

Assim como a seleção e o cruzamento, existe mais de uma forma de realizar a mutação em um algoritmo genético, tanto o estado da arte quanto este trabalho usam operadores de mutação personalizados, porém existe um ponto em comum entre eles: é a ideia de que existe uma taxa de mutação P_m, que corresponde a possibilidade de ocorrer a mutação em um certo ponto do cromossomo e essa mutação consiste em alterar este valor de uma forma determinada.

Este valor P_m é determinado baseado em experimentação e pela própria natureza do problema, em resumo, a mutação tem um importante papel ao ser responsável por adicionar explorabilidade ao algoritmo e alterar as soluções em busca de regiões inexploradas, potencialmente com ótimos locais e globais ainda não descobertos.

2.1.5 Elitismo

Dentro do contexto de algoritmos genéticos existe o fenômeno denominado Elitismo, quando um AG utiliza elitismo, uma quantidade de indivíduos não toma parte do processo normal de seleção, cruzamento e mutação. Estes indivíduos são mantidos intactos até a próxima geração, formando uma elite inalterada. Esta elite pode ser delimitada por meio de um percentual relativo à população total ou a um tamanho fixo (MISHRA; SHUKLA, 2017).

2.1.6 Fluxo Geral

O fluxo geral do Algoritmo Genético pode ser descrito da seguinte forma: O algoritmo irá gerar uma lista de possíveis soluções aleatórias dentro do espaço de busca do problema e irá repetir o processo descrito na figura 1 até que a condição de finalização seja alcançada. A condição de finalização depende do problema e da implementação do algoritmo. (MISHRA; SHUKLA, 2017)

Os operadores do algoritmo genético podem ser implementados de mais de uma forma, podendo existir mais de um algoritmo de seleção ou de cruzamento. Cada algoritmo terá propriedades diferentes que podem ser mais eficientes para uma classe de problemas.

2.1.7 Subpopulações e Migrações

Desde sua concepção, é evidente que o Algoritmo Genético tem potencial para paralelização (BELDING, 1995). Uma das formas de paralelizar o algoritmo genético é conhecido como o modelo de subpopulações ou ilhas, onde o algoritmo genético é executado com várias populações menores com interação restringida entre si.

Cada ilha executa o AG normalmente e busca soluções com sua população reduzida, no entanto, além de existirem várias populações, existe o fenômeno da migração, na qual subpopulações podem ter seus indivíduos removidos e inseridos em outra subpopulação (BELDING, 1995).

Esta migração acontece em períodos fixos e predefinidos, assim como as quantidades de indivíduos trocadas entre subpopulações escolhidas de forma aleatória ou de forma fixa (BELDING, 1995).

2.1.8 Diversidade

Num algoritmo genético, a métrica chamada de diversidade é utilizada para medir o grau de variação ou diferença entre as soluções, ou indivíduos numa população. A diversidade é um conceito importante dentro dos algoritmos genéticos (URSEM, 2002), uma vez que ela ajuda a garantir que o espaço de busca é explorado apropriadamente e que boas soluções não são perdidas pela falta de variabilidade.

Existem várias métricas possíveis para algoritmos genéticos, dentre as quais é possível incluir:

2.1.8.1 Distância de Hamming

Na teoria da informação, a distância de Hamming entre duas palavras de tamanho igual é o número de posições nas quais os símbolos correspondentes diferem (WAGGENER, 1995). Em outras palavras, a distância mede o número de substituições necessárias para

mudar uma palavra e obter a outra, ou o número de erros que poderiam ter transformado uma palavra na outra. Essa métrica é nomeada em homenagem ao matemático americano Richard Hamming.

2.1.8.2 Distância Euclidiana

A distância euclidiana é calculada a partir da distância da reta que conecta dois pontos num espaço multidimensional (SMITH, 2013), onde cada dimensão corresponde ao gene num cromossomo. Pelo seu funcionamento pode ser usado em indivíduos onde a codificação é dada por números reais.

2.1.8.3 Distância de Manhattan

A distância de Manhattan consiste na diferença absoluta entre as coordenadas de dois pontos num espaço multidimensional (KRAUSE, 1986). É similar a distância Euclidiana, porém pode se imaginar a distância Euclidiana como a reta que liga o ponto A ao ponto B, enquanto a distância de Manhattan é a soma dos catetos que tem como hipotenusa essa reta entre A e B.

A escolha da métrica de diversidade depende das características do problema sendo resolvido e do tipo de cromossomo utilizado. Por exemplo, a distância de Hamming pode ser apropriada para uma codificação binária, enquanto a Euclidiana pode ser usada para cromossomos que consistem em um ponto num espaço multidimensional real.

É possível também utilizar mais de uma métrica para alcançar uma visão ainda mais abrangente do problema em questão.

2.2 Busca Aleatória

A busca aleatória pode ser definida como um algoritmo de busca na qual pontos dentro do espaço de busca são escolhidos aleatoriamente até que a condição de término seja alcançada.

Esta condição de término pode consistir em um número máximo de iterações, pontos ou uma qualidade mínima para o ponto encontrado (ROMEIJN, 2009).

2.3 Testes de Software

O mundo moderno é cada vez mais dependente de *Software* e, na medida que essa dependência aumenta, aumenta também a importância de existir confiança nos fragmentos de software das quais se depende. Somente nos Estados Unidos, se estima que todo ano existem perdas de cerca de 59 bilhões de dólares por falhas de software (PLANNING,

2002), cifra maior que o Produto Interno Bruto de alguns pequenos países. Além disso, estima-se que desta cifra, 22 bilhões em perdas poderiam ser evitadas com testes de software conduzidos de forma apropriada (PLANNING, 2002).

Testes de software podem ser definidos como a tarefa de executar um programa ou um sistema para encontrar erros (MYERS; SANDLER; BADGETT, 2011). Também pode se definir testes de software como qualquer atividade cujo objetivo é avaliar um atributo ou capacidade de um programa, ou sistema e determinar se os resultados esperados foram obtidos (HETZEL, 1988).

Diferente de objetos físicos, um software não sofre depreciação com o tempo, por mais que sua infraestrutura possa sofrer, após a sua construção ele é cristalizado na forma que foi construído, apenas as estruturas e seu contexto podem se alterar.

Testes são importantes para validar sistemas ao longo do tempo, porque por mais que um programa não mude, os objetivos do sistema podem mudar. Um exemplo simples é um sistema de cálculo de impostos para uma empresa qualquer, o cálculo do imposto pode ser alterado na lei e o código passa a ser incorreto, mesmo sem nunca ter sofrido modificação alguma.

Erros ou bugs de software vão sempre existir, uma vez que eles nascem não da inaptidão dos programadores ou preguiça, mas porque seres humanos tem um limite para a complexidade que conseguem processar e sistemas tendem a ser complexos, e sua complexidade, intratável (PAN, 1999).

Dada a definição do teste como a execução de um sistema com um resultado esperado, pode se dividir os tipos de teste em dois, explorados nas duas próximas subsecções.

2.3.1 Testes Caixa Branca

Nos testes chamados de Caixa Branca ou Whitebox, os detalhes do sistema são visíveis aos testadores, assim, estes detalhes podem ser usados para diminuir a complexidade dos testes. É possível se basear na própria estrutura do código e do que se espera que cada parte do todo faça para delinear um plano de testes, este tipo de teste é por vezes chamado de teste de caixa de vidro (MYERS; SANDLER; BADGETT, 2011) ou teste baseado em design (HETZEL, 1988).

Entre os tipos de teste Caixa Branca é possível citar a cobertura de linha, que consistem em tentar usar dados de teste que vão executar trechos de código ao menos uma vez e cobertura de ramos que é similar a cobertura de linha, mas foca em explorar condicionais dentro do código para alcançar todos os ramos e não as linhas específicas (PARRINGTON; ROPER, 1989).

Note que a linha de separação entre testes Caixa Branca e Caixa Preta é tênue e

existem técnicas que podem ser consideradas tanto de Caixa Branca quanto Caixa Preta. Por exemplo, um teste por cobertura de linha pode ser considerado Caixa Preta se ele não tem conhecimento do sistema, apenas das linhas alcançadas com certos dados, já que mais linhas são "melhores"que menos linhas, o método pode ser utilizável mesmo sem informações prévias.

2.3.2 Testes Caixa Preta

Nos teste Caixa Preta ou *Blackbox* as informações internas do sistema não são visíveis ao teste, desta forma o teste se baseia nas entradas e saídas do sistema (PERRY, 1989), por vezes este método é chamado de teste baseado em requerimentos (HETZEL, 1988).

É importante que um sistema tenha não apenas um tipo de teste, mas uma mistura de testes diferentes combinando técnicas Caixa Branca e Caixa Preta.

2.4 Testes Automatizados

Testes automatizados consistem no processo de utilizar ferramentas especializadas e técnicas para testar a funcionalidade e desempenho de uma aplicação de software sem a necessidade de intervenção manual. É uma prática importante de manutenção de qualidade que ajuda a garantir a estabilidade e segurança de produtos de software.

Existem vários tipos de testes automatizados: testes de unidade, testes de integração, testes de sistema e testes de aceitação.

2.4.1 Testes de Unidade

Testes de unidade consistem em testar unidades individuais ou componentes de um software para verificar que estão funcionando corretamente. É tipicamente realizado por um desenvolvedor que faz parte do processo de desenvolvimento para capturar erros numa fase mais inicial do desenvolvimento.

2.4.2 Teste de Integração

No teste de integração, se verifica a integração de unidades de software ou componentes e seu funcionamento conjunto. É executada após os testes de unidade para garantir que o software é funcional e segue a especificação.

2.4.3 Teste de Sistema

No teste de sistema, o software na totalidade é testado para verificar se o comportamento segue a especificação. É tipicamente executado após a integração.

2.4.4 Teste de Aceitação

O teste de aceitação consiste em verificar se o software atende aos requisitos e expectativas do usuário final. Normalmente é a última etapa no processo de validação do software.

2.5 Teste Baseado em Busca

Podemos definir Teste Baseado em Busca ou Search Based Testing (SBT) como a utilização de técnicas de otimização e de busca como Algoritmo Genéticos e outras meta-heurísticas para automatizar total ou parcialmente tarefas de teste. Podemos citar como exemplo a geração de dados de teste utilizando o Algoritmo Genético, um ponto-chave para esse processo de otimização é uma função de avaliação específica para o problema, ditando quão bem uma solução soluciona um determinado problema. O objetivo é encontrar soluções viáveis num tempo limite prático dentro do contexto de um espaço de busca potencialmente infinito.(MCMINN, 2011).

A primeira publicação sobre o que hoje é conhecido como Search Based Testing foi publicada em 1976, o trabalho de dois pesquisadores americanos: Webb Miller e David Spooner. A técnica desenvolvida pelos dois autores consiste em gerar testes constituídos por uma sequência de números reais ou um vetor de valores reais, esta técnica era radicalmente diferente das comumente utilizadas em seu período, que eram baseadas em execução simbólica e resolução de restrições. Nessa abordagem, dados de teste eram obtidos por meio da execução de uma versão do código sendo testado, onde essas execuções eram guiadas na direção dos dados desejáveis por meio de uma função de custo (neste trabalho e em outros mais recentes denominada função de avaliação ou função *Fitness*) e um processo de otimização (MCMINN, 2011).

Resultados que eram mais próximos do caminho desejado teriam um valor de custo menor associado, dados com um custo maior eram descartados. Miller e Spooner não continuaram seu trabalho com geração de dados, este trabalho só foi estendido em 1990 por Korel(KOREL, 1990). Desde então houve uma explosão de aplicações na área, não se limitando a apenas gerar dados de teste (MCMINN, 2011).

A área é extremamente ampla e pela própria definição engloba uma série de técnicas e problemas aos quais estas técnicas podem ser aplicadas. Por esse motivo, é preferencial atacar certas subcategorias de problemas ao invés de focar no conceito genérico.

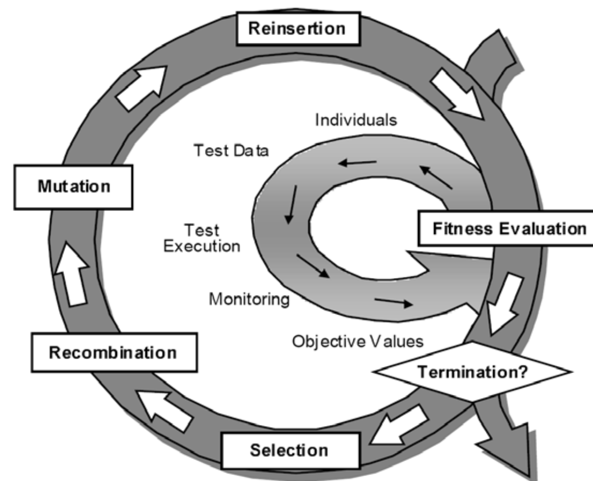

Figura 3. Ilustração do Teste Evolutivo retirado de (SCHLINGLOFF; VOS; WEGENER, 2023)

O foco deste trabalho é gerar dados que serão utilizados em testes de unidade, semelhante ao trabalho inicial de Miller e Spooner, descrito anteriormente.

2.6 Teste Evolutivo

Pode se definir Teste Evolutivo ou *Evolutionary Testing* como uma técnica de teste de software que usa algoritmos evolutivos para gerar e refinar soluções a problemas de engenharia de software (WEGENER; GROCHTMANN, 1998), um conceito similar ao SBT, porém limitado a algoritmos evolutivos como os Algoritmos Genéticos. O objetivo, similarmente ao SBT, é descobrir erros no sistema sendo testado.

A ideia básica por trás do Teste Evolutivo é de usar os princípios de seleção natural para guiar o processo de busca em direção a casos de teste ou entradas efetivos. A população inicial é gerada aleatoriamente assim como no AG e a partir dai as soluções são refinadas até que a condição de finalização seja alcançada, esta podendo diferir, a depender do programa ou algoritmo evolutivo.

Uma das vantagens do Teste Evolutivo é a sua capacidade de detectar casos de erro sem necessitar de intervenção humana a todo momento, esta é uma propriedade dos algoritmos evolutivos, uma vez que é necessário apenas definir a função *fitness* e as condições de contorno do problema e a solução é encontrada de forma consideravelmente automática. Pela sua própria definição, o Teste Evolutivo abrange uma grande quantidade de possíveis soluções e possíveis problemas.

Como exposto nos trabalhos relacionados, uma série de estudos utilizou o Teste Evolutivo com sucesso em uma série de problemas. Um dos motivos de sua utilização é que

os métodos evolutivos, em especial o AG, tem características de busca mais globais, tendo sido utilizado bem em métodos de otimização (WEGENER; GROCHTMANN, 1998).

Pode se observar na Figura 3 a ilustração do processo de teste evolucionário, os operadores descritos nesta imagem serão melhor explorados na próxima secção, onde o algoritmo genético será descrito, uma vez que este é o algoritmo principalmente utilizado no teste evolucionário.

2.7 Teoria dos Jogos

A teoria dos jogos é um ramo da matemática que estuda a tomada de decisão estratégica em situações onde o resultado depende da ação de múltiplos indivíduos ou grupos. A teoria dos jogos providencia uma forma de analisar situações nas quais os tomadores de decisão afetam uns aos outros (MYERSON, 1991). Essa teoria pode ser aplicada para uma série de problemas como problemas da economia, ciência politica e biologia (JÄGER, 2008).

Os elementos básicos da teoria dos jogos são os jogadores, as estratégias disponíveis para cada jogador e os pagamentos ou *Payoffs* associados a cada resultado possível do jogo (MYERSON, 1991). Jogadores são as entidades que tomam decisões neste jogo, podem ser indivíduos ou grupos. As estratégias serão definidas posteriormente e o *Payoff* consiste no resultado obtido para um jogador no jogo de forma relativa aos outros jogadores.

Podem existir, dentro da teoria dos jogos, jogos de natureza cooperativa e não cooperativa. Nos jogos cooperativos, os jogadores podem realizar acordos e reforçar estes acordos por meio de contratos ou instituições, esta restrição força um comportamento de forma externa ao jogo. Já em jogos não cooperativos, os jogadores dependem apenas do seu próprio interesse e *payoff* para decidir que ações tomar no futuro.

Dentro da teoria dos jogos, existem alguns conceitos importantes, como o equilíbrio de Nash. Este equilíbrio é um estado do jogo no qual nenhum jogador consegue melhorar seu *payoff* ao alterar unilateralmente sua estratégia, em outras palavras, é um estado estável no jogo no qual cada jogador usa a melhor estratégia possível ao considerar as estratégias dos outros jogadores (OSBORNE; RUBINSTEIN, 1994).

Outro conceito importante da teoria dos jogos é a eficiência de Pareto, neste estado nenhum jogador consegue um *Payoff* melhor sem causar uma diminuição de *Payoff* em outro jogador, a eficiência de Pareto é um conceito importante no estudo de jogos cooperativos (OSBORNE; RUBINSTEIN, 1994).

2.7.1 Estratégia e Estratégia Dominante

A estratégia no contexto da teoria dos jogos é definida como uma função que associa cada histórico de ações de todos os jogadores a uma ação para um jogador (OSBORNE; RUBINSTEIN, 1994). Este histórico de ações consiste simplesmente nas ações tomadas pelos jogadores ao longo do jogo.

Considere um jogo com os jogadores $P1$ e $P2$ e o histórico de ações $H1$, uma função que associe o histórico de ações $H1$ a ações para o jogador $P1$ seria uma estratégia para o jogador P1.

Além disso, a estratégia dominante para $P1$ pode ser definida como a estratégia na qual, para um dado jogador, a sua ação é a melhor independente da ação dos outros jogadores (OSBORNE; RUBINSTEIN, 1994).

2.7.2 Dilema do Prisioneiro

O jogo do Dilema do Prisioneiro ou *Prisoner Dilemma*(PD), é um jogo dentro da teoria dos jogos no qual existem apenas dois jogadores. O jogo pode ser descrito da seguinte forma: dois membros de uma gangue são presos. Cada prisioneiro está em uma solitária sem nenhuma forma de conversar ou trocar mensagens com o outro, a polícia admite que não tem evidências suficientes para condenar o par na principal acusação contra eles.

O plano é sentenciar ambos a dois anos de prisão numa acusação inferior, ao mesmo tempo, o policial oferece a eles um acordo, no qual eles devem confessar da acusação principal em troca de proteção contra a acusação menor, assim o outro será preso por 5 anos sob a acusação principal (OSBORNE; RUBINSTEIN, 1994).

Desta forma existem três cenários possíveis: no primeiro nenhum deles confessa e desta forma ambos têm dois anos de prisão, no segundo cenário um confessa e outro não, desta forma o que confessa não cumpre pena alguma e o que não confessa recebe a pena máxima. No terceiro e ultimo caso, ambos confessam e recebem a pena principal, tendo cinco anos de prisão.

2.7.3 Pagamento ou *Payoff*

Na Tabela 1 é possível observar os valores de *Payoff* para cada ação possível considerando os dois agentes do jogo. Pode se observar que os valores são concretos para facilitar a análise, porém, é trivial definir -10 como S, -5 como P, -2 como R e T como 0 assim transformando os valores em abstratos.

De modo geral, $T > R > P > S$, onde $R > P$ implica que a cooperação mútua é uma estratégia superior à delação mútua. Simultaneamente, como $T > R$ e $P > S$ é

Tabela 1. Tabela de Pagamentos do Jogo Dilema do Prisioneiro

A \ B	B não confessa	B confessa
A não confessa	-2 / -2	-10 / 0
A confessa	0 / -10	-5 / -5

possível afirmar que a delação é a estratégia dominante para ambos os jogadores.

O operador de mutação implementado neste trabalho é baseado na efetuação de um jogo deste tipo, utilizando estes valores de *Payoff* para adicionar dinamismo no algoritmo.

2.8 Estado da Arte - Algoritmo Genético DaimlerChrysler

O estado da arte de teste evolutivo, o AG conhecido como DaimlerChrysler ou por vezes Sistema DaimlerChrysler tem aplicações bem estudadas ao SBT (HARMAN; MCMINN, 2009), este pode ser definido como um AG que contém 6 subpopulações de tamanho igual, denotadas por seus índices que vão de 0 até 5.

2.8.1 Mutação no DaimlerChrysler

Cada posição no cromossomo sofre mutação com probabilidade $p_m = 1/len$ onde len é igual ao tamanho do cromossomo para o problema em questão.

A equação 2.1 descreve como calcular o novo valor de um cromossomo se a mutação ocorrer, o valor α_x é 1 com probabilidade 1/16 e 0 de outra forma.

$$z_i = x_i \pm domain_i * 10^{-p} * \sum_{x=0}^{15} \alpha_x * 2^{-x} \tag{2.1}$$

2.8.2 Seleção no DaimlerChrysler

A seleção é realizada por meio de ranking linear. Essa seleção envolve organizar a soluções da população baseada no seu *Fitness* de forma que a primeira solução tem a maior chance de ser selecionada e a última tenha a menor, mas que essa percentagem não seja igual ao *Fitness* percentual dessa solução.

Em outras palavras, mesmo se a primeira solução tiver um *Fitness* tão grande que ele é maior que a soma de todas as outras soluções, sua probabilidade de ser selecionado não vai ser maior que 50%. A probabilidade efetiva de cada solução ser selecionada é previamente determinada por uma distribuição desse ranking linear em particular. Dessa

forma, soluções mais aptas tem maiores chances de serem solucionadas sem viciar a seleção nestes indivíduos com valores altos.

2.8.3 Cruzamento no DaimlerChrysler

O cruzamento é realizado por um método chamado de *Discrete Recombination*, onde a solução filha tem a mesma probabilidade de receber um cromossomo de ambos os pais para uma posição qualquer. Reinserção é feita utilizando elitismo onde os 10% melhores pais e 90% dos melhores filhos sobrevivem para constituir a próxima geração.

2.8.4 Migrações no DaimlerChrysler

As subpopulações são também classificados de acordo com um ranking linear e a cada quatro gerações as populações com menos progresso perdem indivíduos para as com mais progressos, o progresso é baseado neste ranking linear e atualizado a cada iteração baseado na equação 2.2.

$$progress_g = 0.9 * progress_{g-1} + 0.1 * rank \tag{2.2}$$

A população é rebalanceada de forma que os tamanhos das populações reflitam o tamanho dos seus progressos. Adicionalmente, a cada 20 gerações, 10% de toda a população em todas as subpopulações são trocadas aleatoriamente.

Todas as parametrizações do AG DaimlerChrysler são definidas pelos próprios autores dos trabalhos que originaram o mesmo (HARMAN; MCMINN, 2009), não sendo definido pelos autores.

2.9 Trabalhos Relacionados

O SBT existe como um tópico de pesquisa desde 1976 (MCMINN, 2011) e o AG foi aplicado com sucesso a ele em diversas áreas.

Em (JATANA; SURI, 2020) o Algoritmo Genético foi comparado ao algoritmo *Particle Swarm Optimization* (PSO) na tarefa de teste de mutação ou *Mutation Testing* e neste estudo os algoritmos foram comparados em termos de quais alcançam os mesmos resultados em um menor tempo, é concluído que uma versão do AG modificada pode resolver problemas com mais eficiência que um Algoritmo Genético padrão.

O trabalho aqui apresentado é diferente do (JATANA; SURI, 2020) uma vez que é feita a mesma comparação, porém, utilizando um novo operador de mutação.

Em (BÜHLER; WEGENER, 2008), estudos de caso são realizados aplicando um algoritmo evolutivo à tarefa de geração de testes funcionais, especificamente testes de

interação entre componentes de um sistema automotivo, gerando bons resultados por facilitar a geração de casos de teste relevantes para este tipo de problema.

Já em (HARMAN; MCMINN, 2009) o estado da arte em termos de algoritmos genéticos é comparado com o método de Hill Climbing, além disso, é feita uma análise de como as propriedades do AG podem torná-lo mais eficiente em gerar certos tipos de dados. Um método híbrido é apresentado, contendo características globais de busca baseados no AG e características locais baseadas no Hill Climbing.

No trabalho desenvolvido pelos autores são tratados apenas algoritmos genéticos e não híbridos, mas a comparação é útil ao demonstrar que existe uma fundamentação pela qual o AG tem um bom desempenho, e não somente evidências empíricas.

No estudo desenvolvido em (GROSS; FRASER; ZELLER, 2012), o AG é utilizado para gerar dados de teste que sofrem menos com falsos alarmes quando comparados a métodos existentes. Neste estudo, duas formas de gerar testes foram comparadas pelos autores e na primeira, chamadas de código são realizadas de forma Caixa Branca ou *Whitebox* para alcançar a maior cobertura e encontrar erros em fragmentos de código.

Na abordagem evolucional, ações na interface do programa são combinadas em soluções e o AG é utilizado para otimizar essas soluções de modo a maximizar a cobertura. O resultado é composto por dados de teste que geram mais erros reais que podem ser enfrentados por um usuário.

Em (TRACEY et al., 2000) os métodos de *Simulated Annealing* e o AG são utilizados com sucesso para a geração de dados de teste com o fim de gerar exceções.

O objetivo é aumentar a confiança do desenvolvedor em um trecho de código, seja ao demonstrar que uma exceção é difícil de alcançar ou ao descobrir que tipos de exceções podem se esperar de um dado programa. Note que não se pode mostrar que não existem exceções porque o AG não garante que todo o espaço de busca é explorado, assim como muitas técnicas de otimização.

Em (PENTA et al., 2007), o AG é utilizado para gerar dados e configurações para serviços, com o intuito de encontrar falhas nos níveis de serviço acordados ou *Service Level Agreements* (SLA). O trabalho ilustra como o AG pode ser flexível e gerar resultados satisfatórios em áreas distintas.

Já em (ALSHAHWAN; HARMAN, 2011), algoritmos de busca são utilizados para diminuir em 30% os recursos de teste e aumentar a cobertura em 54%, o AG não é utilizado neste estudo, porém ele ilustra como a automação pode ser utilizada para geração de testes, diminuindo os recursos utilizados para alcançar um mesmo resultado ou superior.

O objetivo deste trabalho é similar aos citados anteriormente, porém o objetivo é comparar o estado da arte com um AG trivial que utiliza o operador desenvolvido neste

trabalho e avaliar as diferenças em desempenho e convergência, além de desenvolver uma ferramenta onde estes testes podem ser gerados. Várias aplicações podem ser abrangidas pelo SBT, e por consequência existem vários estudos que aplicam diversos algoritmos na geração de dados.

2.10 Resumo do Capítulo

No capítulo 2 é realizado o embasamento teórico do trabalho, onde foram apresentados todos os conceitos necessários aos capítulos seguintes e a plena compreensão dos trabalhos desenvolvidos posteriormente.

Inicialmente foi definido o que é um algoritmo genético, qual seu funcionamento e seus componentes, além de ser descrito como avaliar sua diversidade, também é apresentado o algoritmo de busca aleatória, utilizado na obtenção e avaliação dos resultados deste trabalho. Em seguida são definidos os conceitos de Teste de Software e Teste Automatizado.

São também detalhados os conceitos de Teste Baseado em Busca e Teste Evolutivo, essenciais a aplicação do algoritmo elaborado neste trabalho ao problema proposto, é realizada uma breve introdução a Teoria dos Jogos, conceito fundamental para compreender o operador de mutação desenvolvido neste trabalho.

O estado da arte de teste evolutivo é apresentado e detalhado e, por fim, os trabalhos relacionados são levantados e detalhados.

3 Biblioteca para Gerar Casos de Teste Automaticamente com Algoritmos Genéticos

Neste capitulo será descrita a biblioteca desenvolvida neste trabalho, além disso, será definido que tipo de funções são suportadas pela ferramenta, como elas devem ser instrumentadas, como os testes são gerados e como os resultados são processados.

O objetivo da biblioteca desenvolvida neste trabalho é o de gerar casos de teste de forma automática para funções de programação na linguagem Python, portanto, a própria linguagem Python foi escolhida para o desenvolvimento desta ferramenta, o que facilita a interoperabilidade e a facilidade de uso da biblioteca.

É importante destacar que o objetivo da biblioteca é economizar tempo de desenvolvimento para testes de software, dessa forma, as etapas manuais foram projetadas com o objetivo de facilitar a implementação.

3.0.1 Funções Suportadas

Neste trabalho, o escopo das funções suportadas é limitado a funções onde as entradas podem ser representadas por meio de números reais. Isto inclui letras e texto, uma vez que uma letra na codificação UTF-8 é apenas um número e um texto é uma sequência de letras.

A entrada deve ter uma dimensão finita predefinida, dessa forma uma entrada de tipo texto deve ter uma quantidade predeterminada de caracteres, por exemplo, um texto de 30 letras é equivalente a uma entrada de 30 números reais, no estudo de caso é posteriormente apresentada uma função que tem como entrada um texto de tamanho fixo, e demonstra a viabilidade desta transformação de entrada.

Uma segunda limitação é que a função a ser utilizada deve ser idempotente, ou seja, executar esta função várias vezes com a mesma entrada resulta nas mesmas saídas. Adicionalmente, a função também não deve alterar suas computações parciais segundo o ambiente, as funções devem ser computações puras independentes de entradas do usuário e contatos com a rede.

Essas são as duas limitações que uma função deve cumprir para ser instrumentada e utilizada pela ferramenta de geração de testes desenvolvida neste trabalho.

3.0.2 Instrumentação das Funções

A instrumentação da função consiste em determinar quais são os pontos do código que se quer alcançar com os testes gerados pela ferramenta. Estes pontos devem ser demarcados manualmente assim como as condições para alcançar o ponto.

A Listagem 1 demonstra como é feita a instrumentação de uma função que irá ser posteriormente utilizada para a geração de testes. A variável $TRACER$ é apenas um exemplo, qualquer variável de qualquer nome poderia ser usada para coletar os dados de execução da função. O ponto importante é que uma variável será utilizada para medir as computações intermediárias da função a ser testada, com o objetivo de determinar quão próximo se chega de alcançar certo nó.

No fim da Listagem 1, é possível observar a criação de objetos do tipo *Branch*, se trata de um nó na representação interna da ferramenta, cada nó representa um ponto de código para o qual será gerada uma entrada de teste que satisfaça todos os requisitos para alcançá-lo. Ou seja, o objetivo é gerar uma sequência de entradas que alcance todos os nós.

Não é necessário que todo algoritmo use as informações providas por esta instrumentação, no entanto, muitos algoritmos de otimização necessitam da informação provida por esta instrumentação para gerar os casos de teste.

Será aprofundado em como o Algoritmo Genético utiliza estes dados de instrumentação para gerar os casos de teste posteriormente, mas é possível afirmar que esta instrumentação fornece informações que descrevem o quanto uma certa entrada chegou próximo de alcançar certo ponto no código, informação que será útil para qualquer algoritmo que deseja criar entradas que de fato alcancem qualquer ponto do código.

Listagem 1. Função instrumentada

```
def rgb_to_hls(r, g, b):
    maxc = max(r, g, b)
    minc = min(r, g, b)
    sumc = (maxc+minc)
    rangec = (maxc-minc)
    l = sumc/2.0
    TRACER.add("mx-mc", maxc - minc)
    TRACER.add("l", l)
    if minc == maxc:
        return 0.0, l, 0.0
    if l <= 0.5:
        s = rangec / sumc
    else:
        s = rangec / (2.0-sumc)
```

```
    rc = (maxc-r) / rangec
    gc = (maxc-g) / rangec
    bc = (maxc-b) / rangec
    TRACER.add("r-mx", r - maxc)
    TRACER.add("g-mx", g - maxc)
    if r == maxc:
        h = bc-gc
    elif g == maxc:
        h = 2.0+rc-bc
    else:
        h = 4.0+gc-rc
    h = (h/6.0) % 1.0
    return h, l, s

rgb_to_hls_traces = [
    [
        Branch("mx-mc", BranchCond.EQ, 0, 1.0)],
    [
        Branch("mx-mc", BranchCond.NEQ, 0, 1.0),
        Branch("l", BranchCond.LTE, 0.5, 0.5)],
    [
        Branch("mx-mc", BranchCond.NEQ, 0, 1.0),
        Branch("l", BranchCond.GT, 0.5, 0.5)],
    [
        Branch("mx-mc", BranchCond.NEQ, 0, 1.0),
        Branch("r-mx", BranchCond.EQ, 0.0, 1.0)],
    [
        Branch("mx-mc", BranchCond.NEQ, 0, 1.0),
        Branch("r-mx", BranchCond.NEQ, 0.0, 1.0),
        Branch("g-mx", BranchCond.EQ, 0.0, 1.0)],
    [
        Branch("mx-mc", BranchCond.NEQ, 0, 1.0),
        Branch("r-mx", BranchCond.NEQ, 0.0, 1.0),
        Branch("g-mx", BranchCond.NEQ, 0.0, 1.0)],
]
```

3.0.3 Otimizadores

Dentro da biblioteca desenvolvida é definido o conceito de otimizador, este pode ser descrito como um método ou algoritmo pelo qual casos de teste serão gerados para uma dada função instrumentada.

O primeiro passo na construção do otimizador é a definição de uma função de avaliação ou *Fitness* pela qual será possível determinar a qualidade de um determinado caso de teste. A instrumentação pode auxiliar ao desenvolvedor do algoritmo, ao fornecer dados que descrevem o quão próximo um caso de teste chega de um determinado nó ou de avançar na direção de um nó desejado.

Além disso, o otimizador deve ser capaz de iterativamente melhorar seus casos de teste, isso é, para uma função o otimizador deve ser capaz de melhorar progressivamente os casos de teste e alcançar cada vez mais nós.

Uma vez definida a função de iteração e de avaliação, este algoritmo pode ser fornecido à biblioteca desenvolvida em conjunto com a função instrumentada e de forma automática os testes serão gerados para esta função.

3.0.4 Processamento de Resultados

Após a execução do otimizador para geração de casos de teste para uma função instrumentada, o resultado gerado é uniformizado para qualquer algoritmo na ferramenta desenvolvida. O resultado consistirá de informações sobre o algoritmo: gerações até a solução ótima, tempo de execução e diversidade das soluções, além de informações relacionados ao problema: os casos de teste gerados em si.

Os dados sobre o algoritmo podem ser usados para avaliar diferentes métodos na ferramenta, assim como gerar os dados relevantes para a análise do algoritmo aqui desenvolvido.

3.1 Mutação Baseada em Interação Social

O algoritmo desenvolvido neste trabalho consiste em um operador de mutação baseado no GASI, aplicado para a geração de casos de teste para funções Python, quando utilizado em conjunto com um algoritmo genético.

GASI, como proposto por (PEREIRA et al., 2020) é um algoritmo que insere uma fase de interação social no Algoritmo Genético, como um método de controlar a pressão da seleção. Neste trabalho, esta fase de interação social é utilizada como inspiração para o desenvolvimento de um operador de mutação, aqui denominado de Mutação Mediada por Interação Social ou MMIS.

Esse operador de mutação e todos os descritos neste trabalho são aplicados a problemas nos quais as soluções potenciais, portanto, os indivíduos são sequencias de números reais, codificadas como um vetor de números em Python.

Por exemplo, numa função que aceita dois números reais como entrada, a população consistiria de uma série de duplas de números reais, cada um representando uma possível solução dentro do espaço de busca.

Na fase de mutação, a taxa de mutação é utilizada para determinar se uma mutação ocorre em cada variável de cada solução, se a mutação ocorrer, o operador baseado em (PEREIRA et al., 2020) vai iniciar e ele é dividido em duas fases descritas a seguir.

3.1.1 Fase de Interação Social

Na fase de interação social, cada indivíduo toma o lugar de um jogador no jogo do dilema do prisioneiro, onde o primeiro individuo joga este jogo com o segundo, o terceiro com o quarto e assim sucessivamente.

Para que este jogo ocorra, é necessário que cada jogador siga uma estratégia, esta estratégia é designada aleatoriamente para cada índice da população, no início do algoritmo.

As quatro estratégias possíveis são descritas a seguir:

- A solução vai sempre tentar cooperar.
- A solução sempre irá trair.
- A solução tem como primeiro jogada cooperar, em seguida cópia a última estratégia realizada pelo oponente.
- A solução irá cooperar com 50% de chance e terá a mesma chance de não cooperar.

3.1.2 Utilização do Valor de *Payoff* na Mutação

Como explicado no capítulo 2, todo o jogo irá gerar um valor associado a cada indivíduo denominado *Payoff* ou P que representa como um participante desempenhou no jogo em relação aos outros participantes.

$$Z = R \times \frac{P \times (DMAX - DMIN)}{2} \tag{3.1}$$

Na equação 3.1 pode se observar os valores DMAX e DMIN que correspondem aos valores máximos e mínimos do domínio na variável em que a mutação ocorre, a variável R consiste em um valor aleatório tomado de uma distribuição uniforme com média 0 e

desvio padrão igual a um, por fim, P foi definido anteriormente como o fruto da fase de interação social das soluções.

Quando uma mutação ocorre em uma variável de um individuo, existem cinco cenários possíveis:

- O valor desta variável no indivíduo se torna DMIN.
- O valor desta variável no indivíduo se torna DMAX.
- O valor desta variável no indivíduo se torna o valor de outra variável no mesmo individuo escolhida de forma aleatória.
- O valor desta variável no indivíduo é arredondado
- O valor desta variável no indivíduo é somada ao valor computado Z.

As taxas que governam estes processos são respectivamente chamadas de *floor rate*, *ceil rate*, *copy rate*, *round rate*. O quinto cenário não tem uma taxa correspondente, pois ele ocorre quando os outros quatro não ocorrem, ou seja, com probabilidade $1 - floor_rate - ceil_rate - copy_rate - round_rate$.

Essa mutação introduz diferentes magnitudes de movimento numa população que dependem dos resultados de um jogo de DP. Note que o termo R pode ser negativo, então Z pode tanto aumentar como diminuir cada variável.

3.2 Operadores do Algoritmo Genético

No AG desenvolvido neste trabalho, uma seleção de torneio de tamanho igual a cinco foi utilizada. Um dos motivos da escolha deste tipo de seleção é que ela causa a escolha preferencial de indivíduos de alto desempenho enquanto ainda existe a possibilidade de indivíduos inferiores serem escolhidos, comportamento que preserva a exploração no AG.

É importante considerar o tamanho do torneio, um pequeno demais pode fazer com que indivíduos superiores sejam ignorados no processo de seleção, enquanto um valor grande demais pode causar uma pressão grande na direção de indivíduos muito aptos.

Neste trabalho o cruzamento de um ponto foi utilizado, já explicado anteriormente, além disso, uma elite de tamanho 10% foi utilizada.

3.2.1 Função *Fitness*

Para otimizar a geração de casos de teste com algoritmos de busca como o Algoritmo Genético, é preciso ter em mãos uma função *Fitness* bem definida. Como o objetivo é

maximizar a cobertura de código alcançada por uma lista de casos de teste, a métrica a ser maximizada é a cobertura.

Maximizar a cobertura total pode ser um problema complexo, mas é possível decompô-lo em várias tarefas menores ou sub-problemas, ao se dividir cada função a ser testada em vários nós, ou linhas de código e otimizar cada um desses nós individualmente.

Listagem 2. Função exemplo

```
1.  def exemplo(x, y):
2.      if x > y:
3.          if x > 10:
4.              return 0 # No 1
5.          else:
6.              return 1 # No 2
7.
```

Na Listagem 2, pode se observar uma função de exemplo para a qual é possível gerar testes, as funções utilizadas neste trabalho são reais e de código livre, no entanto, esse exemplo está apenas para ilustrar o funcionamento da função *Fitness*, e nos nós utilizados para simplificar o problema.

Note que os trechos comentados são dois, cada função pode ter um ou vários nós dependendo do seu tamanho e complexidade. Nesse exemplo, o primeiro nó é encontrado quando a expressão $x > y$ é verdadeira e $x > 10$. O segundo nó é alcançado quando $x > y$, porém $x \leq 10$. Como cada nó requer uma execução do AG, essa função seria otimizada com duas execuções do algoritmo no mínimo.

Por fim, o *Fitness* é dividido em duas partes e está ilustrado na Equação 3.2: O nível de aproximação e a distância de nó. O nível de aproximação é simplesmente a medida de quantos pontos de controle foram alcançados em direção ao nó destino. No exemplo, se quisermos alcançar o Nó 1, é preciso passar pelas condições $x > y$ e $x > 10$, portanto, para cada condicional satisfeito, o nível de aproximação é incrementado.

$$F = 100.0 * NivelAproximacao + DistanciaNo \tag{3.2}$$

A distância de nó define para um certo ponto o quão próximo ele está de alcançar o próximo, supondo que para uma solução que deseja alcançar o Nó 1, a condição $x > y$ esteja cumprida, o próximo passo é fazer com que $x > 10$ e a distância de nó deve refletir o quão próximo à solução está de alcançar este ponto.

Existem várias equações que podem ser usadas para calcular a distância de nó (MCMINN, 2011), neste trabalho, será usada a equação 3.3. Nesta equação, X é o valor

esperado na expressão, S_x é o valor da expressão, MH é o valor esperado para o maior valor desta expressão, de forma que S_x está sempre entre MH e $-MH$.

$$DistanciaNo = 100.0 - |X - S_x| * \frac{100.0}{MH} \tag{3.3}$$

3.3 Taxonomia do Indivíduo

Cada função a ser otimizada contém uma quantidade diferentes de entrada e portanto a solução em potencial para esta função tem uma composição diferente, no exemplo da listagem 2, existe uma função com dois argumentos, x e y, o individuo para esta população seria uma dupla de número reais. De modo geral, uma função com N argumentos teria uma lista de N números como o individuo básico da sua população. Para que este individuo seja válido e uma solução razoável para o problema basta garantir que os seus valores estão dentro do domínio e restrições do problema.

3.4 Escolha de Parâmetros

A escolha de parâmetros para o Estado da Arte foi baseada no artigo original no qual este algoritmo foi apresentado e desenvolvido, o tamanho da população foi definido como 60 baseado no tempo de execução do algoritmo. A quantidade máxima de iterações foi definida como 250, sendo o ponto no qual os algoritmos avaliados pararam de fazer progresso visível. Tanto o estado da arte quanto o AG com o operador desenvolvido neste trabalho utilizam elitismo com a taxa de 10%, adicionalmente o operador de mutação desenvolvido neste trabalho tem uma taxa de mutação de 12%, floor rate em 12%, ceil rate em 12%, copy rate em 5% , round rate em 4%.

3.5 Resumo do Capítulo

No capítulo 3 deste trabalho é descrita a ferramenta de geração automática de testes, descrevendo desde as funções suportadas e suas limitações até explicações mais detalhadas sobre o processo interno desta ferramenta.

Em seguida, o operador de mutação desenvolvido neste trabalho é apresentado em conjunto com a parametrização realizada para este algoritmo e o estado da arte.

4 Estudo de Caso

O estudo de caso desenvolvido neste trabalho busca validar a ferramenta desenvolvida, além de validar o algoritmo proposto na geração de casos de teste.

Uma questão primordial que se pretende responder com este trabalho é se o operador de mutação desenvolvido pode ser utilizado para problemas de engenharia de software, mais precisamente para o SBT. Para este fim o algoritmo foi comparado com o estado da arte em sua capacidade de alcançar o ótimo em seu tempo de execução, um algoritmo de busca aleatória também foi utilizado para comparar diversas métricas nesse estudo.

Para responder esta pergunta existem três tarefas principais:

- Estabelecer uma base de dados que possa servir de base, com funções que podem ter dados de teste gerados pelo estado da arte de forma comparável ao algoritmo aqui apresentado.
- Instrumentar essa base de dados para ser possível aplicar os algoritmos e, mais importante, compará-los.
- Definir métricas para medir a qualidade da exploração do espaço de busca.

As três secções seguintes detalharão como estas três tarefas foram realizadas neste trabalho, na mesma sequência em que foram apresentadas.

4.1 Geração da Base de Dados

Toda base dados é composta por funções de código aberto encontrados em repositórios públicos de código. O método utilizado para selecionar os códigos consistiu em buscar por códigos com alta visibilidade, e filtrar aqueles que poderiam ser utilizados pelo estado da arte, para que houvesse uma comparação razoável entre os métodos.

O objetivo não é escolher trechos de código por popularidade, mas sim compor a base de dados com fragmentos de código com alto uso que sejam representativos, contendo funções numéricas assim como funções que aceitam texto como entrada.

Diversas funções foram utilizadas e todas elas foram retiradas dos repositórios de código listados na tabela 2, nesta tabela é possível observar na primeira coluna o nome do repositório e na segunda coluna o endereço eletrônico do repositório ou fragmento de código para verificação e reprodutibilidade.

Tabela 2. Repositórios Usados na Base de Dados

Nome do Repositório	URL do Repositório
BinaryOrNot	https://github.com/audreyfeldroy/binaryornot
isbnlib	https://github.com/xlcnd/isbnlib
beziers	https://github.com/simoncozens/beziers.py
Python Spatial Analysis Library	https://github.com/pysal/spopt
wcag-contrast-ratio	https://github.com/gsnedders/wcag-contrast-ratio
colorsys	https://github.com/python/cpython/blob/3.11/Lib/colorsys.py

Tabela 3. Base de Dados de Funções

Nome da Função	Número de Nós
is_binary_string	2
wcag_rgb	2
_linearize	6
yiq_to_rgb	6
rgb_to_hls	5
hls_to_rgb	3
rgb_to_hsv	4
hsv_to_rgb	7
QuadraticBezier._findDRoots	2
QuadraticBezier.tOfPoint	3
CubicBezier.balance	5
isbnlib.canonical	2

Para ser possível realizar analises sobre a base de dados, esta foi instrumentada para que cada função coletada fosse dividida em ramos ou nós que podem ser otimizados individualmente.

4.2 Instrumentação da Base de Dados

Pela definição de função *fitness* utilizada neste trabalho, é perceptível que apenas ter funções de código em mãos não é o suficiente para gerar dados de teste e avaliar sua cobertura, é preciso após ter as funções em mãos definir ramos ou nós manualmente, pontos do código que desejamos alcançar para que cada iteração do algoritmo possa otimizar para nós individuais.

A tabela 3 enumera as funções coletadas ao fim do processo descrito na secção anterior, além de mostrar a quantidade de nós ou ramos que cada função contém.

Após este processo, existem 47 problemas a serem solucionados, cada nó individual necessita uma execução para ser otimizado. Essa coleta seguida da instrumentação das funções constitui a coleta de dados realizada neste trabalho.

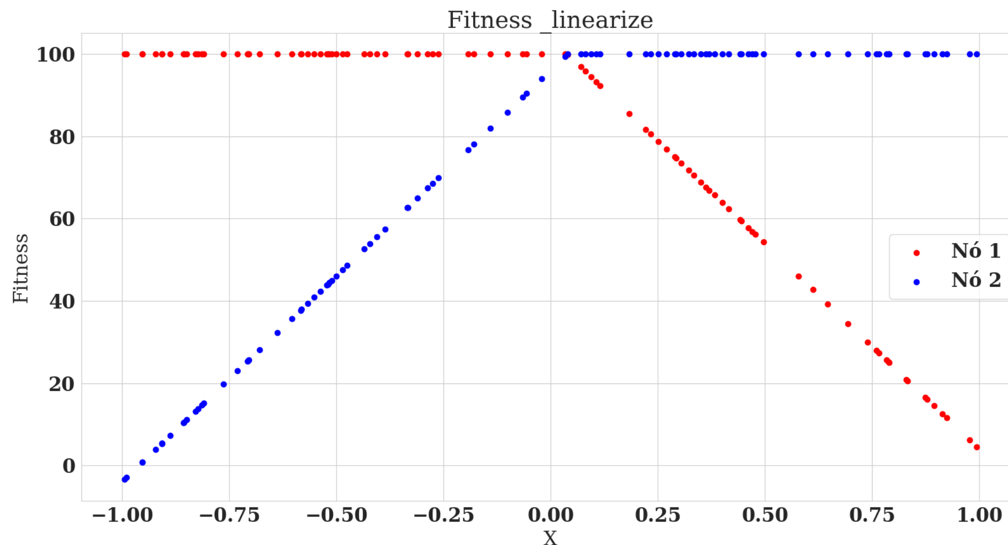

Figura 4. Visualização da função *Fitness* _linearize

4.3 Visualização do *Fitness*

A seguir é feita uma breve análise de algumas funções *Fitness* para exemplificação, duas funções serão avaliadas: _linearize e rgb_to_hsv. As duas foram selecionadas por consistirem em graus diferentes de complexidade, sendo a primeira mais simples e a segunda intermediária.

4.3.1 Função _linearize

É possível verificar na Figura 4 que a função _linearize tem um *Fitness* relativamente simples de navegar, tendo apenas uma dimensão e cada nó tendo valor máximo após certo ponto, funcionando como uma função degrau. Note que cada nó é uma execução diferente do algoritmo, não é necessário encontrar o ponto onde as curvas se encontram, cada curva é uma execução diferente e o *Fitness* é 100 após, ou antes de certo ponto dependendo do Nó.

4.3.2 Função rgb_to_hsv

A função rgb_to_hsv tem como entrada três variáveis, portanto, a visualização é feita por uma sequência de pontos com cores com tons mais vermelhos quanto maior é o *Fitness*. É possível visualizar nas Figuras 5,6, 7 e 8 que existem certas regiões onde o *Fitness* é maior e apesar de dimensionalidade elevada não existem tantos ótimos locais.

Este comportamento da função pode ser explicado pelo fato de que muitos elementos dependem da igualdade entre as entradas, ou da igualdade de valores derivados computados

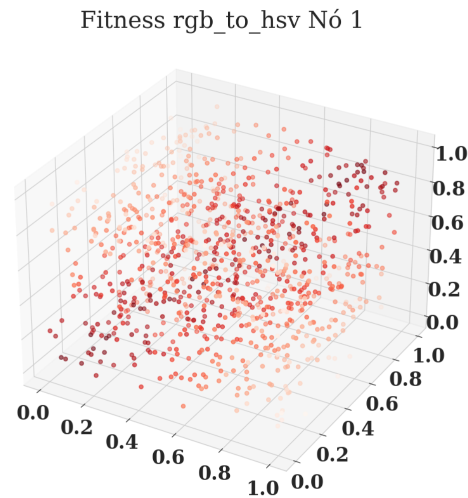

Figura 5. Visualização da função *Fitness* rgb_to_hsv - Nó 1

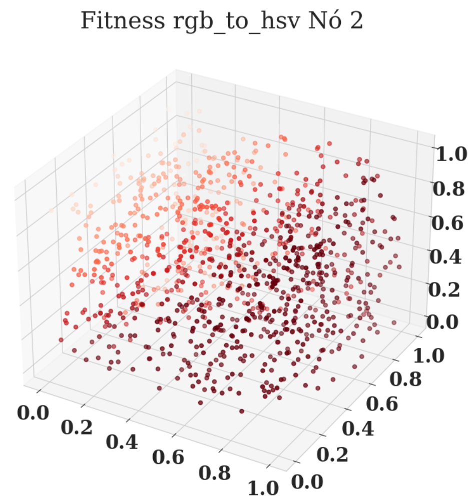

Figura 6. Visualização da função *Fitness* rgb_to_hsv - Nó 2

a partir de operações lineares entre esses componentes, produzindo cortes no espaço de busca onde o *Fitness* é mais elevado, fazendo desta função uma função relativamente simples de otimizar, mas não simples o suficiente para qualquer algoritmo conseguir alcançar o ótimo com consistência.

O objetivo é demonstrar como uma função objetivo se comporta neste trabalho e não mostrar exaustivamente todas as funções objetivo, como todos os problemas otimizados aceitam variáveis reais como entrada, a visualização de cada uma destas funções se daria

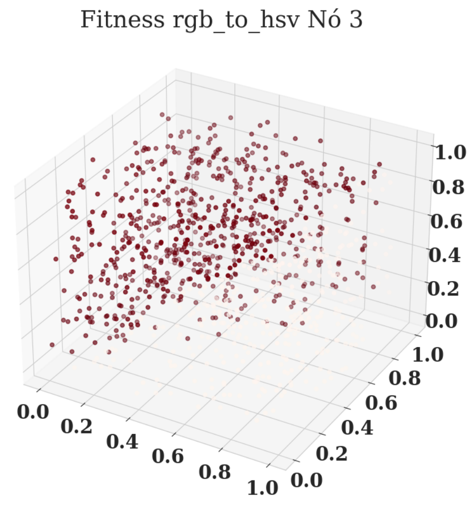

Figura 7. Visualização da função *Fitness* rgb_to_hsv - Nó 3

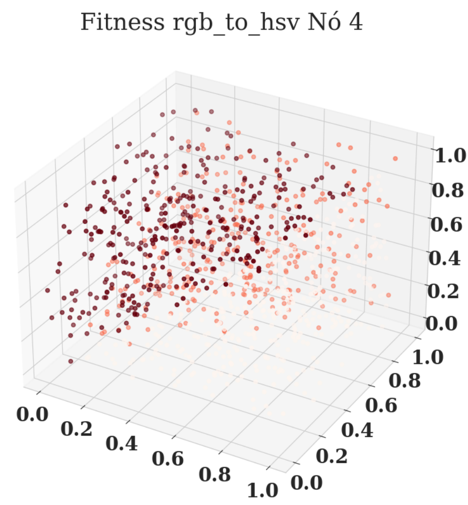

Figura 8. Visualização da função *Fitness* rgb_to_hsv - Nó 4

por meio da visualização multidimensional de números reais, de forma semelhante a como foi exposto nesta secção.

Claramente, existem diferenças de dimensionalidade, alguns problemas contém 20 dimensões de variáveis livres enquanto outros contém apenas uma.

4.4 Metrificação da Diversidade

Para ser possível avaliar a diversidade dos diversos algoritmos utilizados neste trabalho, a distância euclidiana foi usada entre cada membro da população com os outros membros, somadas essas distâncias e calculada a média, é obtida a medida de diversidade.

No caso do sistema DaymlerChrisler, as seis subpopulações foram consideradas uma mesma população com os indivíduos das seis combinadas e a diversidade foi calculada dessa forma. Pode se interpretar essa diversidade como uma medida de dispersão das soluções, já que uma maior distância média implica um afastamento maior entre todos os indivíduos no espaço de busca.

A distância Euclidiana foi escolhida pela sua simplicidade computacional e por se adequar à codificação utilizada neste trabalho. Uma possível expansão do trabalho seria utilizar mais de uma métrica apropriada a este tipo de codificação para ampliar a avaliação dos algoritmos.

4.5 Justificativa

A metodologia utilizada na escolha das funções da base de dados pretende coletar dados que são característicos de códigos encontrados em sistema de produção, não funções de *benchmark*. É objetivo deste trabalho ser resiliente e otimizar funções independentemente de possíveis exceções e, simultaneamente, comportar funções com propósitos diferentes.

A instrumentação destas funções se baseou no estado da arte para trabalhos semelhantes, considerada pelo autor como suficiente para este caso de uso. Apesar do tempo adicional de execução, já que dividir funções em nós aumenta a quantidade de execuções, os resultados são alcançados em tempo razoável e com maior qualidade quando comparado a função *fitness* que tenta otimizar vários nós de uma vez.

Por fim, a diversidade foi calculada de forma simplificada, porém, sem perder as qualidades necessárias a esta métrica na análise deste trabalho. Como os algoritmos aqui apresentados são de valores reais, uma medida de dispersão por distância no plano real reflete muito bem a dispersão, de forma que permite a análise a até a visualização.

4.6 Limitações

Uma das limitações que podem ser levantadas dessa abordagem é a quantidade de funções. Apesar de a quantidade de nós ser 47, se tratam de uma quantidade limitada de funções e cada nó dentro de cada função tende a ser semelhante nesta base.

Uma segunda limitação se dá na forma como a instrumentação foi feita, como descrito anteriormente, essa instrumentação foi feita de forma manual, sendo sujeita a erros para bases de dados maiores, além do trabalho dispendioso de alterar e instrumentar todas as funções apresentadas.

4.7 Resumo do Capítulo

No capítulo 4, é apresentado o estudo de caso realizado com a ferramenta de testes desenvolvida neste trabalho.

É detalhado o banco de dados de funções utilizadas e a instrumentação das mesmas, após isso, é feita uma análise destas funções e da medida de diversidade adequada a elas. Por fim, é levantado a justificativa e as limitações desta escolha de base de dados.

5 Resultados Obtidos

5.1 Ambiente de Programação e Detalhes de Hardware

Como explicado no capítulo 4, as funções utilizadas são funções de código aberto desenvolvidas em Python, desta forma o próprio ferramental e o AG foram também desenvolvidos em Python para avaliar estas funções.

Cada função foi otimizada 20 vezes por cada algoritmo, o hardware utilizado para agregar os resultados é composto por um processador Intel(R) Core(TM) i5-9400 com frequência de 2.90GHz e seis núcleos físicos, além de uma memória RAM de 16 GB e 2400 MHz de frequência. O processamento não utilizou a GPU para processamento, apenas o processamento normal de uma aplicação em Python, utilizando apenas uma *thread*, sem paralelismo.

5.2 Algoritmos Comparados

É esperado que ao menos dois algoritmos sejam comparados em sua performance para a base de dados deste trabalho, o estado da arte e o AG com o operador de mutação proposto neste trabalho. Nos gráficos apresentados neste capitulo também são adicionados os dados obtidos para o algoritmo de busca aleatória, para servir de comparação como um algoritmo sem grande complexidade.

5.3 Cobertura Encontrada

Na Figura 10 e na Tabela 4, pode se observar que os valores de *Fitness* convergem de forma mais acelerada com o uso da MMIS e que o Algoritmo DaimlerChrysler não parece convergir para 100% em todos os algoritmos, essas médias significam que o DaimerChrysler não conseguiu alcançar a convergência em algumas execuções, mas não significa necessariamente que ele nunca alcançou o ponto ótimo para certas funções.

A função _linearize tem um nó onde a cobertura só estará completa se uma variável toma valores extremos, rgb_to_hsv contém condicionais que comparam duas variáveis numa de uma solução. Ambas essas funções, assim como outras, criam um equilíbrio delicado possivelmente solucionado pela MMIS por meio de seu operador de mutação.

Apesar de demonstrar uma cobertura superior nestes casos, o objetivo deste trabalho não é apresentar um valor alto de *Fitness* como uma demonstração de qualidade superior da MMIS, até porque esse tipo de exposição não melhora o estado da arte (HOOKER,

Tabela 4. Resultados Tabelados

Função	**DaimlerChrysler Iterações**	**MMIS Iterações**	**DaimlerChrysler Cobertura**	**MMIS Cobertura**
is_binary_string	5.275	**1.925**	100.0	100.0
wcag_rgb	1.0	1.0	100.0	100.0
_linearize	11.416	**1.0**	95.83	**100.0**
yiq_to_rgb	4.1	**1.5583**	100.0	100.0
rgb_to_hls	7.56	**1.83**	100.0	100.0
hls_to_rgb	1.4	**1.0**	100.0	100.0
rgb_to_hsv	20.725	**2.275**	98.75	**100.0**
hsv_to_rgb	1.071429	**1.0**	100.0	100.0
QuadraticBezier._findDRoots	1.0	1.0	100.0	100.0
QuadraticBezier.tOfPoint	1.0	1.0	100.0	100.0
CubicBezier.balance	1.04	**1.01**	100.0	100.0
isbnlib.canonical	62.45	**5.75**	98.33	**100.0**

1995). Porém, é possível concluir por meio destes dados de que a MMIS pode ser tão efetiva quanto ou até mais efetivo que o estado da arte para uma seleção de problemas de geração de dados para testes.

Pode ser argumentado que um algoritmo de busca com características globais vai ser sempre capaz de encontrar o ótimo global eventualmente, porém outro aspecto importante dos resultados é a velocidade de convergência. Esta velocidade pode aumentar a utilidade de ferramentas geradas a partir desse tipo de busca, se a diversidade é mantida e a busca é saudável, uma convergência mais rápida pode não ser um problema para o usuário final.

Pode se observar pela figura 9, que a diversidade se manteve a níveis altos durante toda a execução do programa. Além disso, a convergência foi mais rápida na MMIS e tomou menos gerações em média, isso é relevante como já apontado pela possibilidade de uso deste tipo de busca em uma ferramente, a geração de testes de unidade como uma ferramenta para engenheiros de software deve ter a possibilidade de ser rápida e responsiva, simultaneamente, deve ser possível existir uma longa execução para gerar uma suíte de testes otimizada. Para usos mais imediatos e de experimentação com código, no entanto, um tempo de execução mais curto para soluções aceitáveis é importante.

5.4 Diversidade

Os níveis de diversidade podem ser observados na Figura 9, pode ser confuso visualizar a MMIS como tendo maior diversidade quando comparado a busca aleatória, isso pode ser explicado pelo fato de que a MMIS tem uma chance de mover soluções radicalmente dentro do domínio e diferente do DaimlerChrysler isso se mantém constante ao longo da execução.

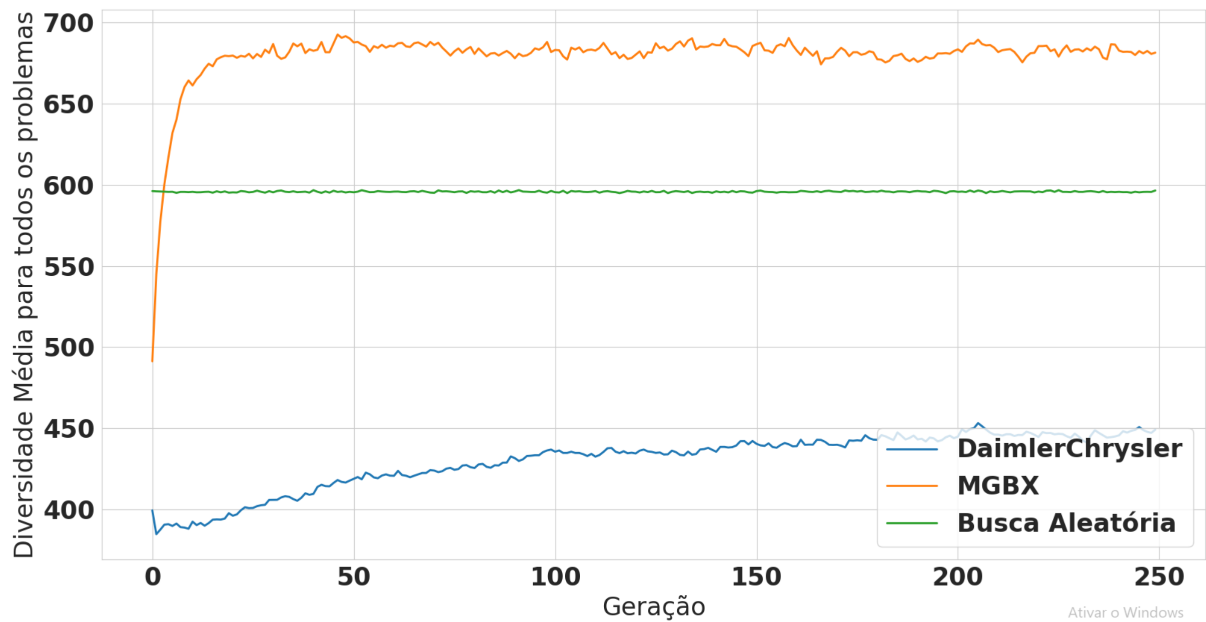

Figura 9. Diversidade por Algoritmo

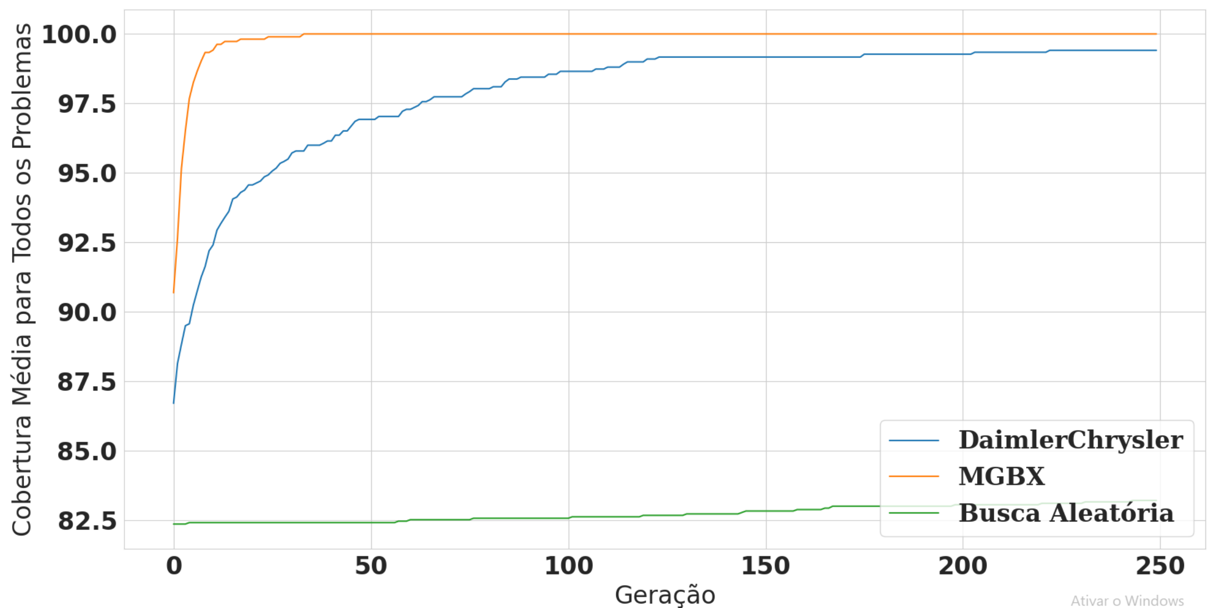

Figura 10. Cobertura Média Alcançada por Algoritmo

5.5 Velocidade de Convergência

Pode se observar na figura 10 que a cobertura converge rapidamente tanto no DaimlerChrysler quanto na MMIS, sendo mais rápida no segundo, é possível observar também que apesar da convergência mais rápida o ótimo é encontrado e considerando a figura 9 pode se perceber que a diversidade se mantém elevada.

Na Figura 11 é possível observar o número médio de gerações até o ótimo em cada

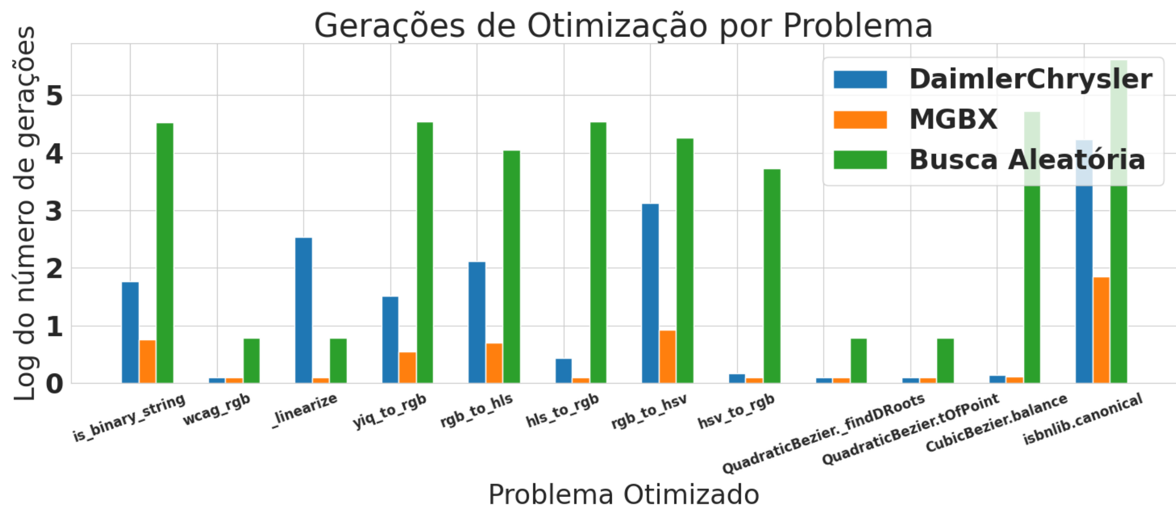

Figura 11. Média de Gerações Até o Ótimo

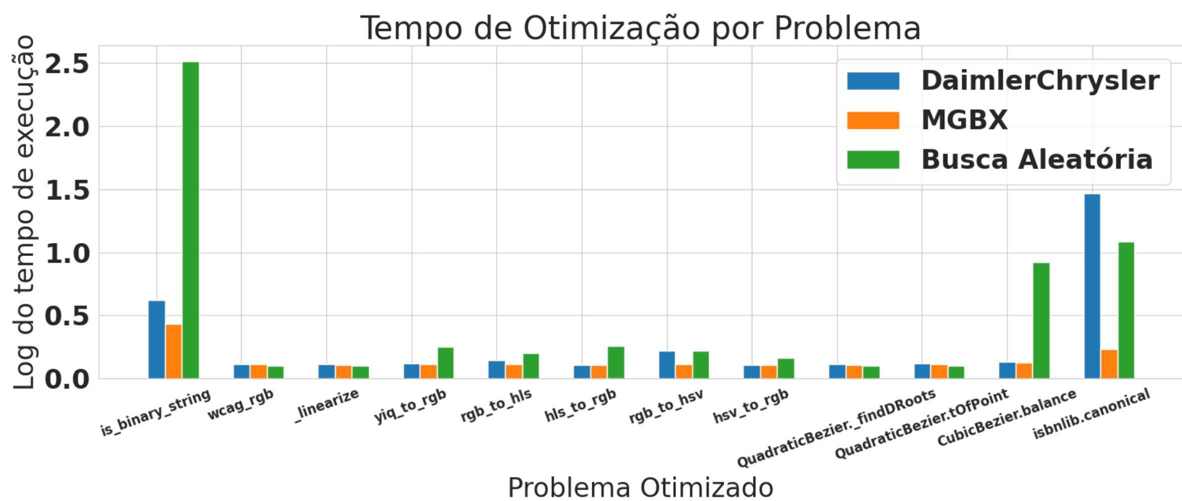

Figura 12. Tempo Despendido por Algoritmo por Problema

algoritmo, note que a MMIS teve uma menor quantidade de iterações até o ótimo.

Analisar somente o número de iterações não é suficiente, é possível observar na figura 12 que o tempo real de execução também é menor para a MMIS, pode se concluir com isso que a menor quantidade de gerações não é explicado por uma maior complexidade do algoritmo em cada iteração, muito pelo contrário. Note que neste gráfico, quando o ótimo não é encontrado, o valor considerado é o máximo possível de iterações.

Na figura 12 é possível observar mais distorções no tempo total no DaimlerChrysler, pois ele inclui seis subpopulações, migrações e, em geral, toma mais segundos por iteração, como confirmado pelo gráfico.

5.6 Resumo do Capítulo

No capítulo 5 são expostos os resultados obtidos pelo estudo de caso, inicialmente o ambiente de testes é descrito, em seguida é demonstrada a cobertura encontrada pelos testes gerados pela ferramenta. A ferramenta consegue gerar dados de teste que alcançam uma cobertura aceitável utilizando tanto o estado da arte (DaimlerChrysler) quanto o algoritmo genético com a operação de mutação proposta neste trabalho (MMIS).

Em seguida, a diversidade alcançada pelos algoritmos é comparada, assim como a cobertura alcançada e o tempo de convergência.

6 Considerações Finais

O estudo desenvolvido neste trabalho demonstrou que um operador de mutação baseado em interação social foi aplicado ao problema de geração de dados de teste com sucesso, além de ter sido demonstrado que a ferramenta desenvolvida consegue gerar os casos de teste com sucesso moderado, alcançando bons resultados na maioria das funções instrumentadas e avaliadas.

Existem dois pontos principais ao se analisar o desempenho deste operador, primeiramente é importante que a convergência aconteça, uma vez que é preciso que dados sejam gerados para que o usuário final consiga realizar seus testes, em segundo é preciso que o tempo de processamento não seja alto, ou seja, que o AG possa gerar resultados bons o suficiente em um tempo baixo.

Durante o desenvolvimento deste trabalho foi exposto que estes objetivos foram alcançados, o operador proposto além de ser aplicável ao problema, também obteve um desempenho superior ao estado da arte em termos de tempo de execução para uma convergência compatível.

Sobre as diferenças em convergência, pode se dizer que o DaimlerChrysler se comporta de forma previsível em um ambiente controlado, até por ser concebido para otimizações numéricas, porém muito do código aberto utilizado neste trabalho não é matematicamente complexo, porém, contém dependências entre as variáveis e isto tem o potencial de influenciar os resultados quando comparamos algoritmos diferentes neste tipo de base de dados.

6.1 Trabalhos Anteriores

É importante ressaltar que o trabalho apresentado teve desenvolvimentos anteriores, no Apêndice A é exposto o trabalho intitulado "Algoritmo Genético Aplicado à Geração Automática de Casos de Teste"e no Apêndice B é exposto o trabalho intitulado "*Application of a Genetic Algorithm with Social Interaction to Search Based Testing*"publicados nos eventos ERIM e LA-CCI, respectivamente.

A proposta inicial do trabalho amadureceu gradualmente até que fossem alcançados os resultados aqui apresentados.

6.2 Dificuldades Encontradas

Durante o desenvolvimento do trabalho uma série de dificuldades foram encontradas, enumeradas nas seções seguintes:

6.2.1 Instrumentação das Funções

Uma dificuldade técnica encontrada foi a de coletar as informações referentes aos nós visitados por cada entrada para cada função, uma vez que essa coleta deve acontecer sem aumentar significativamente o tempo de execução da função, quando comparada a uma execução normal.

Esta etapa foi implementada manualmente neste trabalho, como citado anteriormente, de forma que não onerasse a execução normal do algoritmo. Isto é importante, uma vez que a avaliação ou computação do *Fitness* é repetida diversas durante a geração dos casos de teste, sendo importante que sua execução seja a mais breve possível.

Uma ferramenta comum no ecossistema Python, *Coverage.py*, foi ponderada para esta tarefa sem sucesso, o desempenho alcançado por essa ferramenta foi muito inferior ao esperado, possivelmente por operações de disco realizadas pela mesma para registrar dados de cobertura.

Dado este problema de desempenho e a pouca flexibilidade dessa ferramenta, em conjunto com a falta de opções de código aberto, foi desenvolvido uma forma manual de metrificação que consistiu em instrumentar as funções para as quais os dados de teste estão sendo gerados, como exposto anteriormente no trabalho.

6.2.2 Construção da Base de Dados

Dado que cada função precisa ser instrumentada manualmente, a construção da base de dados foi uma dificuldade constante no trabalho. Uma expansão na quantidade de funções a serem testadas seria consideravelmente facilitada com uma instrumentação automática destas funções.

6.2.3 Tempo de Execução do Algoritmo

O tempo de execução do algoritmo foi bastante elevada, dificultando a validação da base de dados completa. Apesar das otimizações realizadas, ainda existem outras otimizações possíveis. Os algoritmos aqui apresentados foram todos executados de forma sequencial, para avaliar seu desempenho e eficiência. No entanto, numa aplicação real seria importante aumentar ao máximo o desempenho de geração de casos de teste, uma possível otimização seria paralelizar o algoritmo.

6.3 Trabalhos Futuros

No presente, existem uma série de ferramentas para escrever testes de software, como a ferramenta Hypothesis (MACIVER; HATFIELD-DODDS et al., 2019), no entanto, uma expansão da base de dados utilizado neste trabalho e a comparação e validação de novos métodos podem dar origem a ferramentas ainda mais eficientes.

Como citado no Estudo de Caso, é possível expandir a análise da diversidade ao introduzir métricas adicionais de diversidade, produzindo uma visão mais abrangente do comportamento dos algoritmos.

Um aspecto deste trabalho que pode ser melhorado é a instrumentação das funções testadas, feita manualmente, uma ferramenta automática pode ser desenvolvida para a linguagem Python visando gerar essa instrumentação de forma automática. Assim seria possível gerar uma base de dados maior com relativa facilidade, uma ferramenta deste tipo melhoraria a usabilidade deste trabalho e futuros trabalhos na geração automática de dados de teste.

Além disso, também seria um possível em um trabalho futuro, adicionar avaliações do formato da função *Fitness* para os casos de teste, identificando possíveis similaridades e disparidades entre as mesmas.

Referências

ALSHAHWAN, N.; HARMAN, M. Automated web application testing using search based software engineering. In: IEEE. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. [S.l.], 2011. p. 3–12. Citado na página 28.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An overview of genetic algorithms: Part 1, fundamentals. *University computing*, v. 15, n. 2, p. 56–69, 1993. Citado na página 14.

BELDING, T. C. The distributed genetic algorithm revisited. *arXiv preprint adap-org/9504007*, 1995. Citado na página 18.

BÜHLER, O.; WEGENER, J. Evolutionary functional testing. *Computers & Operations Research*, Elsevier, v. 35, n. 10, p. 3144–3160, 2008. Citado na página 27.

ELLIMS, M.; BRIDGES, J.; INCE, D. C. The economics of unit testing. *Empirical Software Engineering*, Springer, v. 11, n. 1, p. 5–31, 2006. Citado na página 12.

GROSS, F.; FRASER, G.; ZELLER, A. Search-based system testing: high coverage, no false alarms. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. [S.l.: s.n.], 2012. p. 67–77. Citado na página 28.

HARMAN, M.; MCMINN, P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, IEEE, v. 36, n. 2, p. 226–247, 2009. Citado 5 vezes nas páginas 12, 13, 26, 27 e 28.

HETZEL, B. *The complete guide to software testing*. [S.l.]: QED Information Sciences, Inc., 1988. Citado 2 vezes nas páginas 20 e 21.

HOLLAND, J. H. Genetic algorithms. *Scientific american*, JSTOR, v. 267, n. 1, p. 66–73, 1992. Citado na página 14.

HOOKER, J. N. Testing heuristics: We have it all wrong. *Journal of heuristics*, Springer, v. 1, n. 1, p. 33–42, 1995. Citado na página 46.

JÄGER, G. Applications of game theory in linguistics. *Language and Linguistics compass*, Wiley Online Library, v. 2, n. 3, p. 406–421, 2008. Citado na página 24.

JATANA, N.; SURI, B. Particle swarm and genetic algorithm applied to mutation testing for test data generation: a comparative evaluation. *Journal of King Saud University-Computer and Information Sciences*, Elsevier, v. 32, n. 4, p. 514–521, 2020. Citado na página 27.

KOREL, B. Automated software test data generation. *IEEE Trans. Softw. Eng.*, IEEE Press, v. 16, n. 8, p. 870–879, aug 1990. ISSN 0098-5589. Disponível em: <https://doi.org/10.1109/32.57624>. Citado na página 22.

KRAUSE, E. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, 1986. (Dover Books on Mathematics Series). ISBN 9780486252025. Disponível em: <https://books.google.com.br/books?id=IW7ICV0QXWwC>. Citado na página 19.

MACIVER, D. R.; HATFIELD-DODDS, Z. et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, v. 4, n. 43, p. 1891, 2019. Citado na página 52.

MCMINN, P. Search-based software testing: Past, present and future. In: IEEE. *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. [S.l.], 2011. p. 153–163. Citado 4 vezes nas páginas 12, 22, 27 e 36.

MISHRA, A.; SHUKLA, A. Analysis of the effect of elite count on the behavior of genetic algorithms: A perspective. In: IEEE. *2017 IEEE 7th International Advance Computing Conference (IACC)*. [S.l.], 2017. p. 835–840. Citado 2 vezes nas páginas 17 e 18.

MITCHELL, M. *An introduction to genetic algorithms*. [S.l.]: MIT press, 1998. Citado 3 vezes nas páginas 14, 16 e 17.

MYERS, G. J.; SANDLER, C.; BADGETT, T. *The art of software testing*. [S.l.]: John Wiley & Sons, 2011. Citado na página 20.

MYERSON, R. *Game Theory: Analysis of Conflict*. Harvard University Press, 1991. ISBN 9780674341159. Disponível em: <https://books.google.com.br/books?id=1w5PAAAAMAAJ>. Citado na página 24.

OSBORNE, M.; RUBINSTEIN, A. *A Course in Game Theory*. MIT Press, 1994. ISBN 9780262150415. Disponível em: <https://books.google.com.br/books?id=UHv1DAAAQBAJ>. Citado 2 vezes nas páginas 24 e 25.

PAN, J. Software testing. *Dependable Embedded Systems*, v. 5, p. 2006, 1999. Citado na página 20.

PARRINGTON, N.; ROPER, M. *Understanding software testing*. [S.l.]: E. Horwood, 1989. Citado na página 20.

PENTA, M. D.; CANFORA, G.; ESPOSITO, G.; MAZZA, V.; BRUNO, M. Search-based testing of service level agreements. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. [S.l.: s.n.], 2007. p. 1090–1097. Citado na página 28.

PEREIRA, R. L.; SOUZA, D. L.; MOLLINETTI, M. A. F.; NETO, M. T. S.; YASOJIMA, E. K. K.; TEIXEIRA, O. N.; OLIVEIRA, R. C. L. D. Game theory and social interaction for selection and crossover pressure control in genetic algorithms: An empirical analysis to real-valued constrained optimization. *IEEE Access*, IEEE, v. 8, p. 144839–144865, 2020. Citado 4 vezes nas páginas 12, 13, 33 e 34.

PERRY, W. E. *A Standard for Testing Application Software, 1990*. [S.l.]: Auerbach Publishers, 1989. Citado na página 21.

PLANNING, S. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, p. 1, 2002. Citado na página 20.

ROMEIJN, H. E. Random search methodsrandom search methods. In: ______. *Encyclopedia of Optimization*. Boston, MA: Springer US, 2009. p. 3245–3251. ISBN 978-0-387-74759-0. Disponível em: <https://doi.org/10.1007/978-0-387-74759-0_556>. Citado na página 19.

SCHLINGLOFF, H.; VOS, T.; WEGENER, J. 08351 summary – evolutionary test generation. 01 2023. Citado 2 vezes nas páginas 6 e 23.

SHUKLA, A.; PANDEY, H. M.; MEHROTRA, D. Comparative review of selection techniques in genetic algorithm. In: IEEE. *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*. [S.l.], 2015. p. 515–519. Citado 3 vezes nas páginas 14, 15 e 16.

SMITH, K. *Precalculus: A Functional Approach to Graphing and Problem Solving*. Jones & Bartlett Learning, 2013. (The Jones & Bartlett learning series in mathematics). ISBN 9780763751777. Disponível em: <https://books.google.com.br/books?id=ZUJbVQN37bIC>. Citado na página 19.

SRINIVAS, M.; PATNAIK, L. M. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, IEEE, v. 24, n. 4, p. 656–667, 1994. Citado na página 17.

TRACEY, N.; CLARK, J.; MANDER, K.; MCDERMID, J. Automated test-data generation for exception conditions. *Software: Practice and Experience*, Wiley Online Library, v. 30, n. 1, p. 61–79, 2000. Citado na página 28.

UMBARKAR, A. J.; SHETH, P. D. Crossover operators in genetic algorithms: a review. *ICTACT journal on soft computing*, v. 6, n. 1, 2015. Citado na página 16.

URSEM, R. K. Diversity-guided evolutionary algorithms. In: SPRINGER. *Parallel Problem Solving from Nature—PPSN VII: 7th International Conference Granada, Spain, September 7–11, 2002 Proceedings 7*. [S.l.], 2002. p. 462–471. Citado na página 18.

WAGGENER, W. M. Pulse code modulation techniques. Springer Science & Business Media, 1995. Citado na página 18.

WEGENER, J.; GROCHTMANN, M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, Springer, v. 15, n. 3, p. 275–298, 1998. Citado 2 vezes nas páginas 23 e 24.

A Artigo ERIM

Algoritmo Genético Aplicado à Geração Automática de Casos de Teste

Adilson de Almeida Neto[1,2,3], **Rodrigo Lisbôa Pereira**[2,3], **Roberto Célio Limão de Oliveira**[1,2,3]

[1]Programa de Pós-Graduação em Engenharia Elétrica (PPGEE),
Universidade Federal do Pará (UFPA)

[2]Laboratório de Computação Bioinspirada (LCBio),
Universidade Federal do Pará (UFPA)

[3]Laboratório de Tecnologias Computacionais (LabTeC),
Universidade Federal Rural da Amazônia (UFRA)

almeidneto@gmail.com, rodrigo.lisboa@ufra.edu.br, limao@ufpa.br

Resumo. *Testes são uma parte essencial do desenvolvimento de software e, neste sentido, este trabalho propõe um framework para comparação de algoritmos na tarefa de geração de casos de teste e posterior comparação desses resultados para Algoritmos Genéticos, Algoritmos Genéticos com Interação Social e Hill Climbing.*

1. Introdução

Em 1975 foi apresentada a ideia da "adaptação em sistemas naturais e artificiais" [Holland 1975] que fundamentalmente consiste em aplicar os princípios da evolução natural para problemas de otimização. John Holland estudou formalmente o fenômeno da adaptação natural e desenvolveu métodos para que seu mecanismo pudesse ser importado para sistemas de computadores, construindo assim os Algoritmos Genéticos (AG ou GA — *Genetic Algorithm*) [Sonkar et al. 2012].

Desde então, o AG é utilizado como meta heurísticas bioinspirada e vem sendo aplicado em diversos estudos de otimização, nos mais variados campos, inclusive na Engenharia. Neste sentido, este trabalho aplicará AG ao problema de geração de casos de teste, objetivando aumentar a cobertura de testes com os dados gerados. Como proposta, será implementada duas variações de AGs e será realizada uma análise comparativa entre os algoritmos. O primeiro AG será conforme o padrão originalmente proposto por John Holland [Holland 1975] e segundo AG será hibiridizado com Interação Social por meio da Teoria dos Jogos (TJ), visando expandir os estudos propostos em [Pereira et al. 2020].

Além desta introdução, este artigo apresenta na seção 1 a justificatica do trabalho, na seção 3 é apresentado um breve estado da arte, na seção 4 é destacado o escopo definido para o trabalho e, por fim, na seção 5 são apresentadas as considerações finais.

2. Justificativa

De acordo com a Engenharia de *Software*, o teste de *software* trata-se de uma fase crucial em um processo de desenvolvimento, onde ocorre o processo de identicação e transformação de defeitos latentes em ações para serem sanadas, de acordo com uma determinada necessidade identificada no *software*. Apesar da natureza trabalhosa e

custosa, não é possível ignorar a fase de teste no ciclo de desenvolvimento do *software*. Esta fase usa em torno de 50 a 60% dos custos atrelados à produção de *software* [Ramler and Wolfmaier 2006] e, segundo [Sommerville 2011], desempenha um papel crucial em qualquer projeto.

Pela sua necessidade, a fase de teste de *software* é considerada uma boa oportunidade para diminuir os custos da produção e, consequentemente, do tempo de desenvolvimento de um *software*. Apesar da complexidade proveniente por esta etapa, aplicar teste de *software* no ciclo de desenvolvimento provê o aumento da qualidade do produto, desde que seja realizada, corretamente, as boas práticas da Engenharia de *Software* [Sommerville 2011].

3. Estado da Arte

Diversos estudos aplicam a interdiscipliaridade do AG com a Engenharia de *Software*. Todavia, dois estudos são destacados. No primeiro, há a aplicação do AG à geração de testes [Sharma et al. 2016]. E no segundo, há a comparação do algoritmo genético com o algoritmo de otimização Hill Climbing e com outros algoritmos presentes na literatura científica [Harman and McMinn 2010]. É importante frisar que nesses dois estudos há a implementação de um AG padrão e de outro AG associado a outro método matemático ou à outra técnica de otimização, o que vai de encontro às principais motivações deste trabalho.

Uma motivação é refrente ao desenvolvimento de uma ferramenta genérica, que será disponível não apenas como um objeto de um estudo pontual e direcionado a uma determinada problemática, mas que poderá ser utilizada por outros pesquisadores que pretenderem desenvolver estudos correlatos. Outra motivação é alusiva a construção do AG associado com uma técnica que visa prover a melhoria do processo de otimização da metaheurística, através da hibridização com a TJ, que será inserida para realizar a interação social no AG. Posteriormente será realizada uma análise, através do teste de *software*, da sua aplicabilidade em alguns problemas que ainda serão identificados.

4. Escopo do Trabalho

O escopo deste trabalho se dá em três partes. A primeira é referente a construção uma ferramenta de testes (*framework*) onde será possível avaliar os casos de teste para uma função genérica, tendo como métrica de qualidade a cobertura alcançada, ou seja, a quantidade de linhas de código percorridas utilizando os casos fornecidos.

A Figura 1 exemplifica o comportamento do *framework* que será construído. De acordo com a figura, o *framework* aceitará uma função a ser testada e as entradas que serão utilizadas, e retornará a cobertura alcançada por estes testes, assim como o detalhamentoo de informações de onde o código não foi percorrido.

A principal diferença entre esta ferramenta e uma ferramenta de cobertura de código usual é que a mesma além de computar as linhas percorridas, também computará porque certos trechos de código não foram adentrados, ou seja, computará as diferenças de valores esperados para que condicionais relevantes fossem alcançados.

A segunda parte do escopo é referente a implementação de três algoritmos para serem utilizados na ferramenta, sendo:

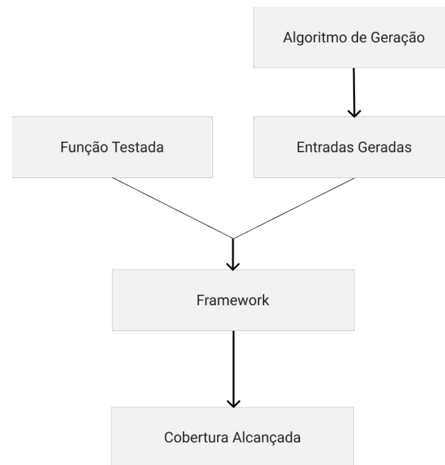

Figura 1. *Framework* de Otimização de Cobertura.

- Algoritmo Genético com Interação Social;
- Algoritmo Genético sem Interação Social;
- Algoritmo Hill Climbing.

A terceira e ultima parte do escopo deste trabalho é referente a avaliação dos três algoritmos, usando como base uma coleção de funções de código livre baseada em [Harman and McMinn 2010], para identificar qual tem a melhor eficácia na geração de casos de teste.

5. Considerações Finais

O trabalho divide-se em três etapas, sendo: 1- Implementação; 2- Definição de problemas; e 3- Análise estatística. Atualmente, o projeto está na etapa 1 e, consequentemente, algumas simulações ainda estão sendo realizadas. Por isso, não foram apresentados resultados. No entanto, os dados da literatura [Harman and McMinn 2010] indicam que esta categoria de abordagem para geração de dados de teste é viável, restando apenas encontrar o equilíbrio entre algoritmos de busca com características mais globais e características locais, e isto será realizado na etapa 2.

Visando a consolidação do estudo, será realizada na etapa 3 uma análise estatística, e através da utilização de algumas métricas indicadas pela literatura, pretende-se corroborar e validar todos os resultados obtidos através das simulações dos algoritmos. Esta etapa irá comparar as métricas de cobertura de teste máximas e médias atingidas à cada problema, em cada algoritmo, para validar a aplicabilidade do algoritmo genético com interação social ao problema e a confiabilidade estatística dos resultados.

Cabe destacar que a aplicação deste trabalho em testes de propriedade é uma área que está sendo estudada pela comunidade científica há alguns anos, onde os algoritmos genéticos já foram usados com sucesso na geração de casos de teste de exceção [Tracey et al. 2000], que podem ser vistos como testes de propriedade onde a propriedade é gerar uma exceção.

Por fim, além da aplicação para a geração dos casos de teste, os algoritmos que serão gerados neste trabalho poderão ser utilizados, também, para a redução de casos de teste baseados em propriedades [Lo et al. 2019]. [MacIver et al. 2019]

Referências

Harman, M. and McMinn, P. (2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36:226 – 247.

Holland, J. (1975). An introductory analysis with applications to biology, control, and artificial intelligence. *Adaptation in Natural and Artificial Systems. First Edition, The University of Michigan, USA*.

Lo, F.-Y., Chen, C.-H., and Chen, Y.-p. (2019). Genetic algorithms as shrinkers in property-based testing. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 291–292, New York, NY, USA. Association for Computing Machinery.

MacIver, D., Hatfield-Dodds, Z., and Contributors, M. (2019). Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4:1891.

Pereira, R. L., Souza, D. L., Mollinetti, M. A. F., Serra Neto, M. T. R., Yasojima, E. K. K., Teixeira, O. N., and De Oliveira, R. C. L. (2020). Game theory and social interaction for selection and crossover pressure control in genetic algorithms: An empirical analysis to real-valued constrained optimization. *IEEE Access*, 8:144839–144865.

Ramler, R. and Wolfmaier, K. (2006). Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. pages 85–91.

Sharma, A., Patani, R., and Aggarwal, A. (2016). Software testing using genetic algorithms. *International Journal of Computer Science & Engineering Survey*, 7:21–33.

Sommerville, I. (2011). *Engenharia de software*. Pearson Prentice Hall.

Sonkar, S. K., Malviya, A., Gupta, D., and Chandra, G. (2012). Software testing using genetic algorithm.

Tracey, N., Clark, J., Mander, K., and Mcdermid, J. (2000). Automated test-data generation for exception conditions. *Softw., Pract. Exper.*, 30.

B Paper LA-CCI

Application of a Genetic Algorithm with Social Interaction to Search Based Testing

Adilson de Almeida Neto[1, 2, 3], Rodrigo Lisbôa Pereira[1, 3], Roberto Célio Limão de Oliveira[1, 2, 3]

[1] Laboratory of Bionspired Computing (LCBIO), UFPA, Belém, Brazil
[2] Post Graduation Program in Electrical Engineering, Federal University of Pará (UFPA), Belém, Brazil
[3] Computational Technologies Laboratory (LabTeC), Federal Rural University of Amazonia (UFRA), Paragominas, Brazil

***Abstract*—Genetic Algorithms are widely used in the Search Based Testing field [1] and in this work, a variation of the Genetic Algorithm with Social Interaction [2] is implemented and applied to the problem of generating software test data. This algorithm is shown to be robust and comparable to the state-of-the-art, having a greater population diversity while converging faster to the global optimal solution.**

***Index Terms*—genetic algorithm, search based testing, game theory**

I. Introduction

Testing software as an activity can take from 30% to 50% of the total effort in software development [3], a reasonable number since lack of proper testing is likely to be the source of software errors [1], causing not only problems in production but costs to stakeholders.

Because of this high cost in productivity for manual testing, research in automatic testing is an important subject [4], tools that can automate the generation of test cases, where applicable, are invaluable.

The creation of one such tool is no small endeavor, a narrow focused solution might become unusable in many problems, while a generic solution can become unable to solve any problem effectively. The flexibility and applicability of GA's to a wide range of problems make it a good candidate for this problem.

The field of Search Based Testing can be defined as the application of search based optimization techniques to the generation of test data [5], the applicability of Genetic Algorithms to this field have been widely studied [1] and in this work we propose the application of a variation of the Genetic Algorithm with Social Interaction (GASI) [2] to the problem of Search Based Testing.

Specifically, we apply GASI to the problem of generating test data. We have tested and compared this algorithm with the state-of-the-art, the DaimlerChrysler system [1], leading to promising results.

An overview of the papers follows: in section II, a brief explanation of the current related research is given, section III exposes the theoretical background needed to understand the results and methods of this work, section IV details the dataset utilized in this research, section V explains the methodology applied, section VI demonstrates the results obtained after experimentation, and lastly, section VII concludes the paper.

II. Related Work

Search Based Testing have several applications, such as reducing false alarms in the generation of test data [6], the automatic generation of test data for web development [7] and testing of Service Level Agreements (SLA) [8].

Search Based Testing (SBT) exists as a research topic since 1976 [4] and Genetic Algorithms have been applied with success to it, in areas such as mutation testing [9], functional testing [10], structural testing [11] and exception testing [12].

Previous literature have also compared the performance of GA with the Hill Climbing algorithm [1] and with the Particle Swarm Optimization algorithm [9].

The objective of this work is similar, but we aim to compare the state-of-the-art with another variation of GA, the Genetic Algorithm with Social Interaction [2], and evaluate the differences in performance.

III. Theoretical Background

A. Unit Testing

Unit testing can be defined as the testing of individual hardware or software units or groups of related units [13], this can be interpreted as a test that validates the result from a single, isolated function [3], care is taken to assert that the functions under test are properly isolated and suffers no influence from external dependencies.

Typically, a test case consists of the testing prerequisites, the test data and the test oracle that decides whether the test passed [14]. In the context of this work, a test case consists of the function to be tested and a list of inputs represented by a Genetic Algorithm individual.

The objective of this work is not to generate data that will pass a unit test, rather, we want to generate data that will maximize code coverage as if the unit test is just executing the function itself.

The applicability of this generated data can be found using other methods, such as property based testing, in which a large set of generated data is applied to a function under test, and some property is checked after each execution [15].

An interesting example is a pair of functions that can compress and decompress data, a property that should be true at all times is that a piece of data should be the same after compression and decompression. This work could be used to

generate data to maximize coverage for both these functions, and the generated data should keep this mentioned property, for example.

B. Social Interaction Based Mutation

GASI, as proposed by [2], is an algorithm that inserts a social interaction phase in the Genetic Algorithm, as a method of controlling selective pressure. In this work, this algorithm was modified so that the social interaction phase is used to control the exploration and exploitation aspects of the algorithm during the mutation phase, this means that the proposed algorithm can be reduced to a new mutation operator, introduced as MGBX.

This mutation operator and all other operators described in this paper are applied to real valued genetic algorithms, in which every solution is encoded as an array of real numbers.

To optimize a function that has two numbers as a parameter, for example, the population would be made of solutions that consist of two real numbers.

In the mutation phase, the mutation rate is used to determine if a mutation occurs at each variable for each solution, if mutation occurs, the mutation operator based on [2] will initiate, and it is divided in two different parts.

In the first part, we use a strategy previously assigned randomly to each candidate solution to play a simulated game of Prisoner's Dilemma - PD with the subsequent candidate solution. Starting at 0, each solution I will play a game of PD with the solution I + 1, the strategies are the following:

1) Always Cooperate: The solution will always try to cooperate.

2) Always Defect: The solution will always try to defect.

3) Tit-for-Tat: The solution will start trying to cooperate and will copy the last seem move from the second move onward.

4) Random: The solution will cooperate or defect with a 50% probability.

To calculate the value of the payoff Z to be used in the next phase, we must use equation 1, where R is a random number taken from a normal distribution with mean 0 and standard deviation of 1, DMAX and DMIN are the maximum and minimum values of the domain. P can be determined by mapping each value of the T, R, P, S payoff rule to a percentage of the domain, in this work, the values used were: 10%, 1%, 0.1%, 0.01%

$$Z = R * \frac{(DMAX - DMIN) * P}{2} \tag{1}$$

In the second and last part, the solution will either be dropped to the minimal value of the domain, raised to the maximum value of the domain, being rounded, having its value copied into another variable or being increased by the Z value from equation 1. The rates governing these events are respectively, the floor rate, the ceil rate, the round rate and the copy rate, increasing by Z will happen when none of the others occur.

This mutation introduces different magnitudes of movement inside the population, depending on the result of a DP game, note that the R term can be negative, so this equation can both increase or decrease the value in each variable.

C. Genetic Algorithm Operators

In this work, the tournament selection was used with tournament size equal to 5, crossover was done using a single point crossover and elitism was also used.

D. Fitness Function

In order to optimize the generation of test cases utilizing search algorithms such as Genetic Algorithms, it is necessary to have a well-defined fitness function. Since our goal is to maximize the code coverage reached by a list of test cases, the coverage is our metric to be maximized.

Maximizing the total coverage can be a complex problem, we can decompose it in smaller sub-problems, splitting any function to be tested into several "branches" or "traces", lines of code we want to reach, and optimizing for one at a time.

Listing 1
FUNCTION TO GENERATE TESTS

```
1.  def example_function(x, y):
2.      if x > y:
3.          if x > 10:
4.              return 0
5.  example_function_traces = [
6.      [Branch(
7.          "x-y",
8.          BranchCond.GT, 0.0, 1000.0
9.      ), Branch(
10.         "x",
11.         BranchCond.GT, 10.0, 1000.0
12.     )]
13. ]
14.
```

In Listing 1, we can observe a sample function which we can generate tests to, while the functions used in this work are real and open source, this is just an example to illustrate the fitness function and the traces used to achieve them.

Note that the "example_function_traces" variable holds a list of branches, these branches are the specific lines of code we want to reach, along with the conditionals that determine if this line of code was reached, depending on the inputs provided.

For example, the first branch is reached when the expression x - y is greater than zero, the last element of the branch is the range, the maximum and minimum value this expression can take, this means x - y can be at most 1000 and is at least -1000.

Each branch needs an execution of the Genetic Algorithm to be optimized, this function contains two branches, so two optimization runs would be necessary.

The fitness is divided into two components: the approach level and the branch distance, the approach level defines how many branches a solution reaches and the branch distance measures how close a solution is to reaching the next branch

on the list, the final value for the fitness can be calculated using Equation 2.

$$F = 100.0 * ApproachLevel + BranchDistance \quad (2)$$

Suppose that on Listing 1, we have x equals to 5 and y equals to 3, the first branch $x - y > 0$ would be reached while the second branch would not, that means our approach level is equal to 1, since one branch was reached.

Branch distance can be usually calculated for various predicates by using equations in [4], for this work, however, we will use Equation 3. In this equation, X is the expected value of an expression, S_x is the actual value of the expression and MH is a "hint" of the range of this expression, so that S_x is always inside the interval between MH and $-MH$.

In other words, in our example, the first branch would give us $X = 0$, $S_x = (5 - 3)$ and $MH = 1000$.

$$BranchDistance = 100.0 - |X - S_x| * \frac{100.0}{MH} \quad (3)$$

E. Overview of the Individual

Another requirement for a Genetic Algorithm is the notion of the individual, the unit of which the population is made of. For this work, each function we generate coverage for will lead to a different population shape, in Listing 1 we have a function with two arguments, x and y, the individual for the population generating coverage to this function would consist of arrays of two real numbers.

F. Parameter Tuning

Parameter tuning applies mostly to the algorithm proposed in this work, since the DaimlerChrysler GA has many of its parameters based on the problem it is applied to.

The maximum number of generations for each problem was determined as 250, this is the required number of generations for both of the algorithms being compared to stop making visible progress.

Both GA's use elitism, and they share the same elite size, 10% population size. Additionally, MGBX has a 12% mutation rate, a 12% floor rate, a 12% ceil rate, a 5% copy rate and a 4% round rate. The population size was set as 60.

G. DaimlerChrysler Genetic Algorithm

Applications of the DaimlerChrysler system to Search Based Testing have been widely studied [1], it can be defined as a Genetic Algorithm with 6 different subpopulations of the same size, denoted by their index P going from 0 to 5.

Each position of the chromosome is mutated with a probability $p_m = 1/len$ where len is equal to the size of the chromosome for the problem.

Equation 4 describes how to calculate the new value for a position of the chromosome if mutation occurs, the value of α_x is 1 with probability 1/16 and 0 otherwise.

$$z_i = x_i \pm domain_i * 10^{-p} * \sum_{x=0}^{15} \alpha_x * 2^{-x} \quad (4)$$

TABLE I
DATASET FUNCTIONS

Function Name	Number of Branches
is_binary_string	2
wcag_rgb	2
_linearize	6
yiq_to_rgb	6
rgb_to_hls	5
hls_to_rgb	3
rgb_to_hsv	4
hsv_to_rgb	7
QuadraticBezier._findDRoots	2
QuadraticBezier.tOfPoint	3
CubicBezier.balance	5
isbnlib.canonical	2

Selection is done through linear ranking, crossover is done with a method called discrete recombination, where a child solution has the same probability of receiving a chromosome position from both parents. Reinsertion is done using elitism, the 10% best parents and 90% of the best children are kept into the new generation.

The subpopulations are themselves ranked linearly and every four generations populations with less progress lose individuals to populations with more progress, progress is based on this linear rank of populations and is described in equation 5.

$$progress_g = 0.9 * progress_{g-1} + 0.1 * rank \quad (5)$$

The population is linearly rebalanced so that population sizes matches progress values. At each 20 generations, 10% of the population in all subpopulations are randomly exchanged.

IV. DATASET USED

A dataset was built for this work using open source Python functions, a total of 12 functions and 47 branches were used, and those branches were manually created for each function.

External dependencies can cause unwanted effects and bias the analysis, because of that, not every fragment of code can be used and filtering took place. Since this work aims to reliably generate unit tests and those should minimize external dependencies, this is a reasonable filter.

An expansion of this dataset is desirable, the Python language was chosen for this work to facilitate contribution and expansion, however, some effort is needed to manually generate all branches. Table I quantifies the functions of the dataset and the number of branches in each.

The intention while creating this dataset was not to create an extremely controlled sample set, but to gather diverse pieces of software, this is why a diversity of functions exists, is_binary_string for instance, uses strings encoded as numbers to generate test cases.

V. METHODOLOGY

To determine the performance of MGBX, both the MGBX and the DaimlerChrysler were executed 20 times for each

TABLE II
AVERAGE ITERATIONS TO OPTIMA AND AVERAGE COVERAGE FOUND BY FUNCTION

Function Name	DaimlerChrysler Iterations	MGBX Iterations	DaimlerChrysler Coverage	MGBX Coverage
is_binary_string	5.275	**1.925**	100.0	100.0
wcag_rgb	1.0	1.0	100.0	100.0
_linearize	11.416	**1.0**	95.83	**100.0**
yiq_to_rgb	4.1	**1.5583**	100.0	100.0
rgb_to_hls	7.56	**1.83**	100.0	100.0
hls_to_rgb	1.4	**1.0**	100.0	100.0
rgb_to_hsv	20.725	**2.275**	98.75	**100.0**
hsv_to_rgb	1.071429	**1.0**	100.0	100.0
QuadraticBezier._findDRoots	1.0	1.0	100.0	100.0
QuadraticBezier.tOfPoint	1.0	1.0	100.0	100.0
CubicBezier.balance	1.04	**1.01**	100.0	100.0
isbnlib.canonical	62.45	**5.75**	98.33	**100.0**

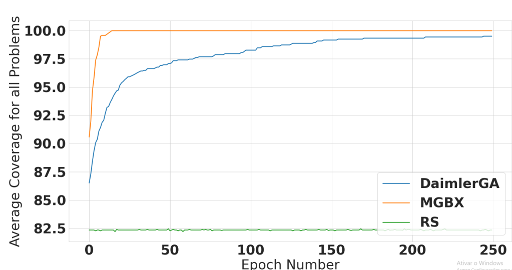

Fig. 1. Average fitness for all problems by generation

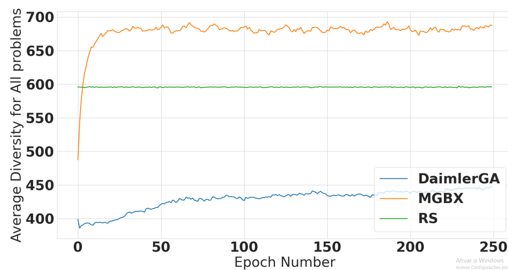

Fig. 2. Average diversity for all problems by generation

problem and the measurements of diversity and best fitness were taken for each generation of each of those 20 executions. The values obtained with these executions will be used to extract conclusions on the Results section.

Population diversity was calculated by using the average euclidean distance inside the population for each generation, both the diversity and the best fitness values by generation were averaged over executions before being saved for further analysis.

To give a frame of reference into the difficulty of the problems, a random search algorithm was also used to benchmark against the GA's, this can give insights into the diversity and performance of both algorithms.

VI. RESULTS

A. Programming Environment and Hardware Details

As explained in section IV, the functions utilized are Open Source Python functions and the algorithms were all implemented in Python.

Each function was optimized 20 times by each algorithm, the hardware used to gather the results had a 2.90 GHz processor with 6 cores and 16 GB of RAM memory, no GPU was used in the processing of the algorithms.

B. Found Coverage

In Figure 1 and Table II, we can observe that the fitness values converge faster with MGBX and that the DaimlerChrysler does not seem to converge to 100% in all algorithms, since those are averages this means that the DaimlerChrysler system failed to converge in some executions, not that it did not converge in any execution for certain functions.

The _linearize function has a branch in which coverage will be complete only if a variable takes extreme values, rgb_to_hsv contains conditionals which compares two variables inside a solution. Both of these functions and others can create a delicate equilibrium that is likely solved on MGBX by using a mix of the copy, ceil, floor and round operators.

While this demonstrate higher coverage for these two cases, the objective of this work is not to present higher fitness values as a demonstration of quality in the algorithm for all cases, as this does not expand the state-of-the-art [16], but rather, to demonstrate that the use of MGBX can be effective in SBT for a demonstrated class of problems.

While it can be argued that a search algorithm with global characteristics will possibly arise at the global optima eventually, the most relevant aspect of this comparison is the speed of convergence of the MGBX.

While diversity stayed at a healthy level, convergence took significant fewer generations with MGBX, this is relevant

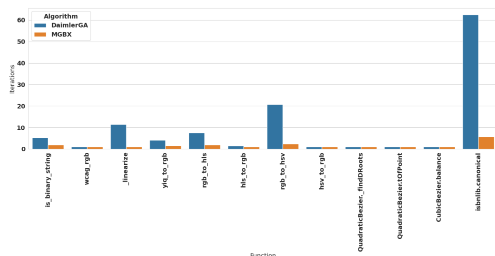

Fig. 3. Number of Iterations Before Optimal Solution

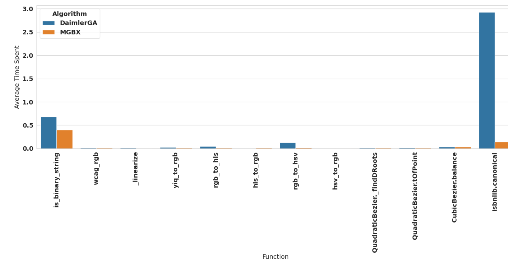

Fig. 4. Execution Time Before Optimal Solution

since generation of unit test cases as a tool used by Software Engineers should be fast and responsive, while they can run for a long time to generate a final suite of tests, if the tool is going to be used in a daily basis to constantly check for issues in code, a faster runtime is of high importance.

C. Diversity

Diversity levels can be observed in Figure 2, it might be confusing to find MGBX to have higher levels of diversity when compared with a random search, this can be explained by the fact that MGBX has a chance of moving solutions closer to the domain boundaries, giving greater average distance between the points. It is also noticeable that the MGBX has greater diversity when compared with DaimlerChrysler, this is an intended effect of the dispersion created by the mutation operator.

D. Convergence Speed

In Figure 3 we can observe the average number of generations to find the optimal test suite for each algorithm for each function, notice that the MGBX took fewer iterations to arrive at the optima.

Analyzing only the number of iterations is not sufficient, in Figure 4 we can also observe that the average total running time before the optimal solutions is lower for the MGBX. Note that if no optimal solution is found, the value is considered as the maximum number of iterations.

Figure 3 has fewer distortions because the DaimlerChrysler system includes 6 subpopulations, migrations and in general takes more seconds per iteration, as the graph itself confirms.

VII. Conclusion

With the use of the MGBX mutation operator, it is possible to achieve results comparable and in some instances surpassing the state-of-the-art while converging faster to global optima.

While controlled environments can benefit from characteristics of DaimlerChrysler, much of the open source code utilized in this work is not mathematically complex, but contains many dependencies between variables, this has the potential to influence any results comparing methods with a similar dataset.

A. Future Work

While the future holds much promise, the present already has a fair share of tools for automating the process of writing software tests [17], an expansion of the dataset used in this work would facilitate the research into methods that can yield even better tools.

Another aspect of this work that can be expanded is the fact that branches for test functions had to be done manually, an automatic tool could be developed for the Python language, creating the possibility of expanding the dataset with ease.

Such a tool would increase radically the usability of this and of future research for unit test case creation.

References

[1] Harman, M. & McMinn, P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions On Software Engineering*. **36**, 226-247 (2009)

[2] Pereira, R., Souza, D., Mollinetti, M., Neto, M., Yasojima, E., Teixeira, O. & De Oliveira, R. Game theory and social interaction for selection and crossover pressure control in genetic algorithms: an empirical analysis to real-valued constrained optimization. *IEEE Access*. **8** pp. 144839-144865 (2020)

[3] Ellims, M., Bridges, J. & Ince, D. The economics of unit testing. *Empirical Software Engineering*. **11**, 5-31 (2006)

[4] McMinn, P. Search-based software testing: Past, present and future.

[5] McMinn, P. Search-based software test data generation: a survey. *Software Testing, Verification And Reliability*. **14**, 105-156 (2004)

[6] Gross, F., Fraser, G. & Zeller, A. Search-based system testing: high coverage, no false alarms. *Proceedings Of The 2012 International Symposium On Software Testing And Analysis*. pp. 67-77 (2012)

[7] Alshahwan, N. & Harman, M. Automated web application testing using search based software engineering. *2011 26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*. pp. 3-12 (2011)

[8] Di Penta, M., Canfora, G., Esposito, G., Mazza, V. & Bruno, M. Search-based testing of service level agreements. *Proceedings Of The 9th Annual Conference On Genetic And Evolutionary Computation*. pp. 1090-1097 (2007)

[9] Jatana, N. & Suri, B. Particle swarm and genetic algorithm applied to mutation testing for test data generation: a comparative evaluation. *Journal Of King Saud University-Computer And Information Sciences*. **32**, 514-521 (2020) *2011 IEEE Fourth International Conference On Software Testing, Verification And Validation Workshops*. pp. 153-163 (2011)

[10] Bühler, O. & Wegener, J. Evolutionary functional testing. *Computers & Operations Research*. **35**, 3144-3160 (2008)

[11] Wegener, J., Baresel, A. & Sthamer, H. Evolutionary test environment for automatic structural testing. *Information And Software Technology*. **43**, 841-854 (2001)

[12] Tracey, N., Clark, J., Mander, K. & McDermid, J. Automated test-data generation for exception conditions. *Software: Practice And Experience*. **30**, 61-79 (2000)

[13] IEEE IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*. pp. 1-84 (1990)

[14] Wappler, S. & Lammermann, F. Using evolutionary algorithms for the unit testing of object-oriented software. *Proceedings Of The 7th Annual Conference On Genetic And Evolutionary Computation*. pp. 1053-1060 (2005)

[15] Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L. & Pierce, B. Foundational property-based testing. *International Conference On Interactive Theorem Proving*. pp. 325-343 (2015)

[16] Hooker, J. Testing heuristics: We have it all wrong. *Journal Of Heuristics*. **1**, 33-42 (1995)

[17] MacIver, D., Hatfield-Dodds, Z. & Others Hypothesis: A new approach to property-based testing. *Journal Of Open Source Software*. **4**, 1891 (2019)