UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS RUSSAS

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

SAMUEL BRITO DA COSTA

APLICAÇÃO DE COLÔNIA DE FORMIGAS PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTES

RUSSAS

2021

SAMUEL BRITO DA COSTA

APLICAÇÃO DE COLÔNIA DE FORMIGAS PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Russas da Universidade Federal do Ceará, como requisito à obtenção do grau de bacharel em Engenharia de Software.

Orientador: Profa. Dra. Jacilane de Holanda Rabelo

RUSSAS

2021

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C875a Costa, Samuel Brito da.
Aplicação de colônia de formigas para geração automática de casos de testes / Samuel Brito da Costa. – 2021.
50 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Engenharia de Software, Russas, 2021.
Orientação: Prof. Dr. Jacilane de Holanda Rabelo.

1. Teste de Software. 2. Meta-heurística. 3. Casos de teste. 4. Automatização. I. Título.

CDD 005.1

SAMUEL BRITO DA COSTA

APLICAÇÃO DE COLÔNIA DE FORMIGAS PARA GERAÇÃO AUTOMÁTICA DE CASOS DE TESTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Russas da Universidade Federal do Ceará, como requisito à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em:

BANCA EXAMINADORA

Profa. Dra. Jacilane de Holanda Rabelo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Pablo Luiz Braga Soares
Universidade Federal do Ceará - (UFC)

Gutemberg Brito Ferreira
Centro Universitário do Norte - (UNINORT)

À todos que me incentivaram e ajudaram durante a trajetória dessa graduação, em especial, minha Família.

AGRADECIMENTOS

Agradeço em primeiro lugar à minha família, em especial minha mãe Clarinda e meu irmão Cláudio por toda a força e incentivo durante essa longa jornada. Sem eles essa conquista não teria sido possível. E os demais familiares: pai, tias, tios, primos e primas que também me apoiaram bastante.

Agradeço à todos os colegas e amigos que fiz durante o curso, com certeza essa caminhada foi abrilhantada por eles. Ao Laelber, Isaac, Jonathan, João Vitor, Rafael, Joaquim, Marcelo, Lucas, Wilker, Fernanda, Rafaelly e todos os outros que me ajudaram e ajudam em tudo. Também a todos os meus amigos fora da faculdade, sempre torcendo e dando total incentivo para concluir essa etapa da minha vida.

Agradeço também aos professores, em especial a Profa. Jacilane e Prof. Pablo que me orientaram nessa pesquisa.

Por fim, agradeço a vida e a oportunidade de poder sempre aprender e melhorar a cada dia.

RESUMO

Uma vez que os *softwares* estão cada vez mais presentes no nosso cotidiano, necessitamos também de produtos de *software* com mais qualidade, que atendam as nossas expectativas em vários aspectos. Para desenvolver um produto com qualidade, os desenvolvedores precisam estar cientes de que a presença de defeitos ou erros impacta negativamente no produto, tornando indispensável a realização de testes, rastreamento a fim de encontrar *bugs* e , quando encontrados, removendo-os para garantir a qualidade almejada. Neste contexto, este trabalho desenvolveu um algoritmo para processar caminhos independentes de um Grafo de Fluxo de Controle que servissem como base para testes estáticos, fomentando a geração automática de casos de teste. Assim, foram levantadas as técnicas utilizadas para geração de um Grafo de Fluxo de Controle e otimização com a meta-heurística Otimização por Colônia de Formigas, auxiliando na resolução deste problema. Os resultados mostram que com a abordagem desenvolvida neste trabalho foi possível encontrar caminhos independentes em funções escritas em *Python*, bem como possibilita que estes caminhos sejam utilizados, em trabalhos futuros, para a geração automática dos valores necessários para testar a função. Conclui-se que o algoritmo desenvolvido foi capaz de gerar casos de testes, podendo auxiliar no desenvolvimento cotidiano, bem como podendo auxiliar nos estudos relacionados à área de testes, com a evolução e melhoria da abordagem desenvolvida.

Palavras-chave: Teste de Software. Meta-heurística. Casos de Teste. Automatização.

ABSTRACT

Living today a daily transformation, as software is increasingly present in our daily lives, the need for higher quality software products becomes greater. To develop a quality product, you need to be aware that the presence of defects or errors negatively impacts the product, making it essential to carry out tests, tracking in order to find bugs and, when found, removing them to ensure the desired quality. In this context, an algorithm was created to process independent paths of a Control Flow Graph that would serve as a basis for static tests, promoting the automatic generation of test cases. Thus, research was carried out on the automatic generation of test data in order to raise the techniques used to read the function to be tested, generate a Control Flow Graph and optimize it with the Ant Colony Optimization meta-heuristic helping to solve this problem. The results show that the approach developed in this work makes it possible to find independent paths to test a function and also allows them to be used, in future works, to generate the defined values to test the function. It is concluded that the developed algorithm was able to generate test data, being able to help in the daily development, as well as being able to assist in the studies related to the test area, with the evolution and improvement of the developed approach.

Keywords: Test of Software. Metaheuristic. Test Cases. Automation.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
ASA	Árvore de Sintaxe Abstrata
CC	Complexidade Ciclomática
CT's	Casos de Testes
GA	Genetic Algorithm
GFC	Grafo de Fluxo de Controle
MoSBaT	Multi-objective Search Based Testing
SA	Simulated Annealing
SBET	Search-Based Energy Testing
SBST	Search-Based Software Testing
SBTSI	Search-Based Test Strategy Identification
VeV	Verificação e Validação

SUMÁRIO

1 INTRODUÇÃO

Vivendo hoje uma transformação diária, a medida em que os *softwares* estão cada vez mais presentes no cotidiano das pessoas, torna-se maior a necessidade de produtos de *software* com mais qualidade (ROSADO, 2020). Para desenvolver um produto com qualidade, precisa estar ciente de que a presença de defeitos ou erros impacta negativamente no produto, havendo assim a necessidade de testes e rastreamento de modo a encontrar *bugs* e removendo-os para garantir a qualidade almejada (ALVES, 2020).

Na garantia da qualidade são exercitadas atividades de monitoramento checando se os padrões e procedimentos acordados estão sendo seguidos, garantindo que os problemas sejam encontrados e as devidas ações ou melhorias dos processos sejam realizadas. Assim, pode-se entender a garantia de qualidade de software como: um padrão planejado das ações pertinentes ao fornecimento de confiabilidade de que um item ou produto está em conformidade com os requisitos e processos especificados (MARTINHO, 2021).

Os testes estão inseridos em um contexto maior da engenharia de *software* conhecido como Verificação e Validação (VeV). Verificação remete às tarefas que asseguram que o *software* implementa, corretamente, uma função especificada. Validação refere-se às tarefas que certificam que o *software* foi desenvolvido conforme os requisitos do cliente (PRESSMAN, 2016).

Como um dos seus grandes benefícios, os testes de *software* permite revelar pontos de falhas para que seja possível corrigi-los antes da entrega do produto final. Com o auxílio de técnicas, é possível desenvolver Casos de Testes (CT's) que são capazes de identificar com mais precisão os pontos de inconformidade, possibilitando sua correção mais brevemente, assim reduzindo custos no projeto em sua totalidade. De acordo com Chiavegatto *et al.* (2013), Souza e Gasparotto (2013), no processo de desenvolvimento de *software* a atividade de testes possui particularidades que dificultam atingir o seu potencial esperado.

Os testes do produto consomem uma boa parcela do esforço de todo o ciclo de desenvolvimento de *software*. Isso enfatiza a necessidade de testes mais rigorosos, com menor custo de tempo, e que consigam encontrar o maior número de falhas para que os produtos sejam entregues no prazo estimado e com alta qualidade. Para isso, muitas empresas estão investindo na automatização das atividades do processo de teste visando o objetivo de melhorar a qualidade do software, aumentar a agilidade do processo de desenvolvimento e reduzir os potenciais riscos na entrega, oferecendo benefícios como: economia de tempo, reuso de *scripts*, incremento de qualidade, precisão e confiabilidade (DIAS, 2018).

Um ponto considerável na atividade de teste é a avaliação da qualidade do conjunto de CT's que será utilizado para testar um *software*, tal que este conjunto conterá os dados necessários para avaliar este *software* em aspectos funcionais e operacionais (DELAMARO *et al.*, 2016). O principal objetivo é utilizar CT's que consigam encontrar o maior número de defeitos desperdiçando o mínimo de tempo e esforço possível (DELAMARO *et al.*, 2016).

Várias pesquisas foram realizadas acerca do assunto, aplicando desde meta-heurísticas até algoritmos de aprendizagem profunda. No trabalho de Arifiani e Rochimah (2016), é proposta uma abordagem que utiliza o Diagrama de Máquina de Estado UML somado ao Teste de Caixa Cinza para a geração de casos de testes e, a partir dessa junção, realizar uma comparação direta com dados gerados em pesquisas focadas somente em Testes Estruturais.

Sharifipour *et al.* (2017) propõem uma variação evolutiva do Ant Colony Optimization (ACO) que visa melhorar a eficiência em cobertura de ramo dos dados gerados automaticamente. Para isso, eles desenvolvem dois pontos principais para atingir os resultados esperados: nova definição de trilhas de feromônio, repassando essa informação para a função de aptidão; e a definição de uma segunda função de aptidão *booleana* para validar novos caminhos independentes.

A proposta deste trabalho é a adaptação da meta-heurística ACO para encontrar os caminhos independentes de um Grafo de Fluxo de Controle (GFC) e a partir destes caminhos fomentar a geração de dados de teste que atinjam as maiores porcentagens de cobertura do mesmo. Dessa forma, espera-se que os caminhos independentes do GFC sejam retornados para servir de base para os testes estáticos da função passada, e assim seja avaliada a eficiência dos testes na verificação da função.

1.1 Objetivos

1.1.1 Objetivo Geral

Propor uma adaptação do *Ant Colony Optimization* para encontrar caminhos independentes de um Grafo de Fluxo de Controle e, a partir destes, fomentar a geração de dados de teste automática viabilizando os testes com caminhos distintos, potencializando a qualidade esperada pela funcionalidade à ser testada.

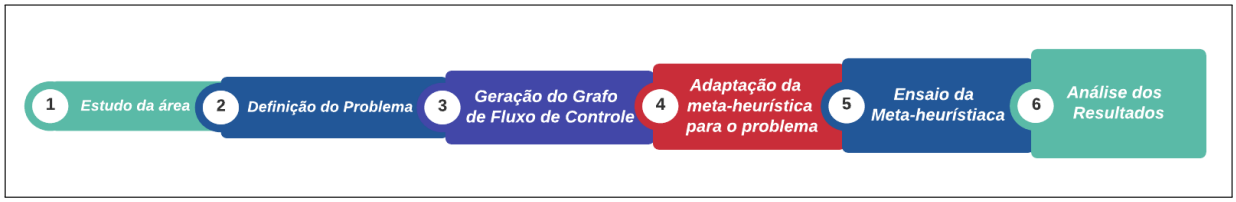
1.1.2 Objetivos Específicos

- Obter, de forma automática, um grafo de fluxo de controle a partir de uma função escrita na linguagem *Python*;
- Obter caminhos independentes do grafo de fluxo de controle aplicando o processamento com o *Ant Colony Optimization*;
- Validar a geração dos caminhos independentes para verificação estática da função;
- Analisar a cobertura dos caminhos independentes validados conforme a função à ser testada;

1.2 Procedimentos Metodológicos

O procedimento metodológico seguido para desenvolver esse trabalho e alcançar os objetivos propostos, está definido nas etapas ilustradas pela Figura 1 e detalhadas em seguida.

Figura 1 – Etapas para aplicação da metodologia



Fonte: O autor (2021)

1.2.1 Estudo da área

De início, foi realizado uma revisão bibliográfica com objetivo de obter um estudo do estado da arte e de identificar as temáticas que englobam a geração automática de casos de teste. Com base nisso, foram definidos diversos temas, como: técnicas de teste, ferramentas de automação de testes, abordagens para geração de dados de testes para que fossem identificadas as principais técnicas e abordagens relacionados ao problema.

Em relação aos trabalhos relacionados, buscou-se por aqueles que tinham como foco a geração de dados utilizando meta-heurísticas, investigando seus objetivos, base de dados utilizadas, métodos adotados e resultados obtidos, buscado-os principalmente no *Google Academic* e em repositórios de congressos e simpósios de qualidade. Para esta etapa, foi utilizada estratégia de busca formulada com as seguintes palavras-chave: *Test Case Generation*, *Metaheurístics*, *Ant Colony Optimization*, *Search-based Software Testing* e *Control Flow Graph and Ant Colony*.

1.2.2 Definição do problema

Após a leitura de alguns trabalhos, foi entendido a aplicação das principais estratégias e abordagens utilizadas para a geração automática de dados de teste e então definido formalmente o problema deste estudo, onde visou-se aplicar a meta-heurística ACO no processo de busca dos caminhos independentes de um GFC, modelado a partir de uma função na linguagem *Python*. Além disso, foi definido a avaliação dos caminhos gerados com a métrica de cobertura de caminhos, na qual a quantidade dos caminhos encontrados em um GFC tem relação direta.

1.2.3 Geração do Grafo de Fluxo de Controle

Na abordagem definida para este estudo, foi desenvolvido um processo de geração de GFC modelando a função à ser testada em um grafo, sendo este, usado como parâmetro para a execução da meta-heurística. O processo consistiu na leitura da função que foi analisada e, a partir da árvore de sintaxe abstrata da mesma, criado um grafo orientado modelando os principais pontos da função, como: declarações, decisões, repetições, etc.

1.2.4 Adaptação da meta-heurística para o problema

Nesta etapa, uma versão básica do ACO aplicada à outro problema (encontrada no *Github*) foi implementado na linguagem *Python*, e então, adaptado para realizar a leitura e o correto processamento de busca dos caminhos independentes no GFC.

1.2.5 Ensaio da meta-heurística

Depois de realizar a implementação e adaptação do ACO, nesta etapa foi feita a execução do processo de busca dos caminhos desejados. O ensaio constituiu-se em gerar o GFC de diferentes funções escritas na linguagem *Python*, mencionadas nos trabalhos relacionados, e então utilizar a meta-heurística para encontrar e retornar os caminhos independentes. Após estes caminhos serem retornados, foram realizados testes criando um conjunto de dados que exercitassem tais caminhos, bem como a validação do processo de geração desenvolvido e análise com base na métrica de cobertura de caminhos.

1.2.6 Análise sistemática dos Dados

Por fim, após o ensaio, foi analisada a eficiência dos dados gerados a partir dos caminhos encontrados pelo ACO com base na métrica de cobertura de caminhos e outras métricas de desempenhos trazidas pelos trabalhos relacionados.

1.3 Organização do Trabalho

Este trabalho esta organizado em 6 capítulos. O Capítulo 1 contextualiza e enfatiza o campo de pesquisa para o problema de geração de casos de teste, apresenta os objetivos gerais e específicos do estudo e, também, apresenta os procedimentos metodológicos. O Capítulo 2 apresenta a fundamentação teórica para a compreensão do ramo que esta pesquisa está inserida. O Capítulo 3 mostra os trabalhos relacionados no qual a pesquisa baseia-se. O Capítulo 4 descreve o desenvolvimento do algoritmo. O Capítulo 5 apresenta os resultados preliminares e definitivos do trabalho. E por fim, o Capítulo 6 apresenta as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta alguns conceitos importantes e fundamentais para a compreensão desta pesquisa. Na seção 2.1 é apresentado o conceito de Teste de Software e suas especificações. Na seção 2.2 e 2.3 são apresentados os conceitos sobre meta-heurísticas e testes de software baseados em busca.

2.1 Teste de Software

Os testes de software, assim como testes de outros produtos como: carros, celulares, dentre outros, tem o objetivo de garantir uma maior qualidade na entrega do produto de software descobrindo falhas no sistema, reportando erros e verificando se os mesmos foram corrigidos.

Existem diversos tipos de testes de software que partem desde os testes de componentes unitários até os testes de aceitação do cliente, onde pode-se citar: Teste de Unidade, Teste de Integração, Teste de Regressão, Teste de Caixa Preta, Teste de Caixa branca etc.

> De uma forma simples, testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado. O objetivo principal desta tarefa é revelar o número máximo de falhas dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos. (J.SILVA *et al.*, 2016)

Para Summerville (2011), os testes de software seguem um processo no qual seus objetivos dividem-se em:

1. Demonstrar ao desenvolvedor e ao cliente que o software atende a seus requisitos;
2. Descobrir as falhas, ou seja, situações em que o software se comporta de maneira incorreta ou indesejada e diferente das especificações.

Assim, fica evidente a necessidade da realização dos testes de software como uma importante atividade para garantia da qualidade do produto desenvolvido, no qual devem ser realizados os procedimentos de teste para a identificação e correção do maior número de defeitos arguindo a operabilidade correta do produto e atendendo aos requisitos especificados.

2.1.1 Tipos de Teste

Em seu livro "A Arte de testar software", MYERS *et al.* (2012) menciona que para confutar os desafios atrelados à economia de teste é necessário estabelecer as estratégias de testes antes de iniciá-los, e ressalva que duas técnicas principais prevalecem: Técnica de caixa-preta e Técnica de caixa-branca. Em complemento, Souza e Gasparotto (2013) relatam as duas técnicas apresentando algumas maneiras de testar o software, visando assegurar seus aspectos gerais ou unitários.

Pressman (2016) defende que uma estratégia de teste de software deve englobar os testes de baixo nível, que verificam se pequenas partes do código fonte foram implementadas corretamente, como também testes de alto nível, que validam as principais funcionalidades do software de acordo com os requisitos dos clientes.

Dentre as técnicas encontradas na literatura, pode-se apresentar:

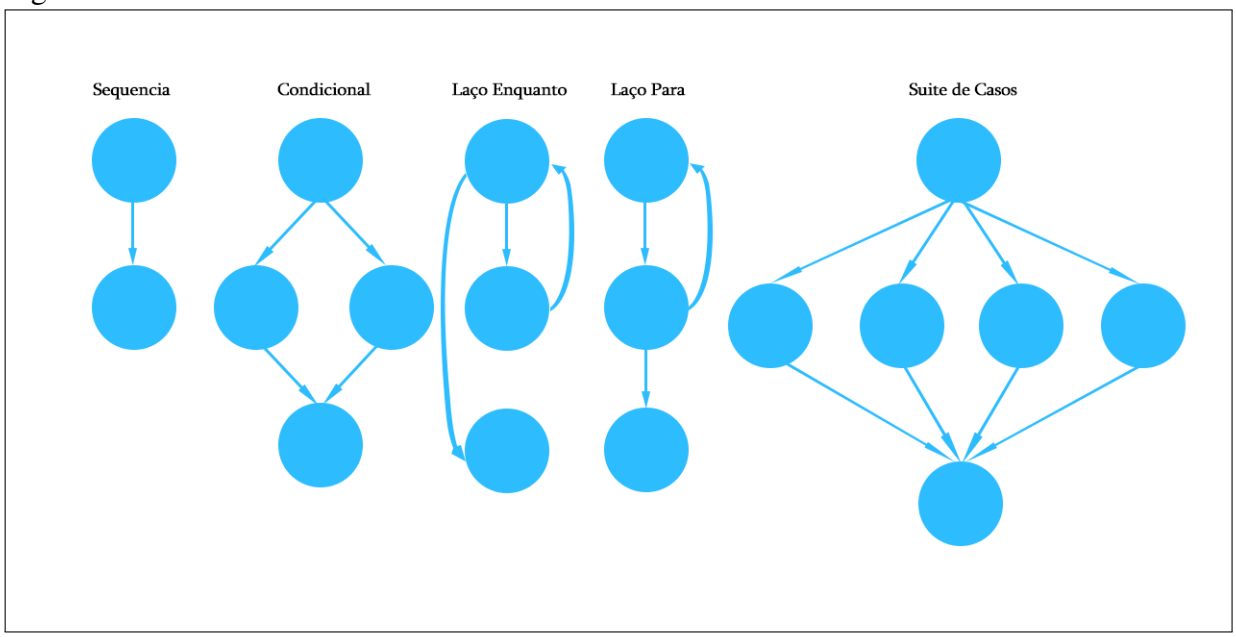
1. **Teste Funcional ou de caixa preta**: é baseado nas funcionalidades do software em alto nível. Este tipo de teste visa garantir que os requisitos foram implementados de acordo com a especificação, onde são verificados a inserção de dados e os resultados retornados sem se preocupar com a estrutura interna do código fonte (PRESSMAN, 2016). Dentre outros, pode-se citar:
    - **Partição de Equivalência:** método que divide o domínio de dados de teste em classes de dados que serão utilizados com base nos possíveis casos de testes;
    - **Análise de Valor Limite:** este método é complementar ao anterior, onde define que serão utilizados os dados das extremidades da partição. Este método deriva casos de teste que verificam a transição de uma classe de dados para outra;

2. **Teste Estrutural ou de caixa branca**: também conhecido como teste estático, é o teste feito na parte interna do software, no código fonte. Tem por objetivo verificação de padrões de desenvolvimento, testar cada procedimento, cada linha, testar os fluxos básicos e alternativos do código-fonte, executar os laços de repetição em suas fronteiras e limites operacionais (PRESSMAN, 2016). A exemplo:
    - **Análise Estática:** é um método de depuração que verifica a qualidade do código-fonte, examinando-o antes da execução. Isso é feito analisando o código em relação a um conjunto de regras de codificação;
    - **Teste de Caminho:** um caminho é o fluxo que será exercitado com base em um condição específica. Dado a complexidade lógica de uma função, é possível derivar casos de testes baseados nos caminhos;
    - **Teste de Estrutura de Controle:** outros testes focados na estrutura de controle aumentam a qualidade dos testes estruturais. Estes testes são especificamente para Condições, Laços de repetição e Fluxo de Dados;
3. **Teste Caixa-Cinza:** dado que o teste funcional não verifica o código da função a ser verificada, a junção do Teste de Caixa Preta e Teste de Caixa Branca nos dão o Teste de Caixa Cinza (LEWIS, 2009). Este método possibilita otimizar o teste funcional com base no conhecimento que o testador terá sobre as estruturas de dados e algoritmos da funcionalidade.

2.1.2 Grafo de Fluxo de Controle (GFC)

O GFC é um grafo que modela a função/parte do código a ser testada, assim modelando o fluxo de controle lógico do programa. Cada aresta representa um ramo da função, e cada nó representa os blocos de comandos onde os ramos da função são definidos pelos comandos de decisão *IF*, *ELSE* ou *SWITCH* (Condicional) e pelas estruturas de repetição *WHILE* ou *FOR* (Laço Enquanto, Laço Para), ilustrados na Figura 2, (VASCO, 2016).

Figura 2 – Estruturas de Controle



Fonte: Adaptado de Pressman (2016)

Para derivar casos de teste com base no GFC, podem-se utilizar de critérios baseados no **fluxo de controle** que definirão os caminhos no grafo. Estes critérios são:

- **Todos os nós:** cada comando do programa (cada nó do GFC) é executado pelo menos uma vez;
- **Todos as arestas:** cada aresta do GFC é exercitada pelo menos uma vez;
- **Todos as condições:** em caso de decisões lógicas, cada premissa deve assumir os valores lógicos verdadeiro e falso.
- **Todos os caminhos:** todos os caminhos possíveis do GFC são executados pelo menos uma vez;
- **Todos os caminhos independentes:** todos os caminhos que introduzem pelo um novo conjunto de comandos. Para o GFC é um caminho que contém um novo ramo ainda não percorrido antes;

Através da meta-heurística ACO foi possível percorrer o GFC utilizando o critério de caminhos independentes, anteriormente mencionado, para encontrar a maior quantidade de caminhos e para auxiliar na geração de dados de teste.

2.1.3 Complexidade Ciclomática

Uma métrica usada para quantificar a complexidade lógica de um programa é a Complexidade Ciclomática. No método de teste de caminho básico, é possível identificar o número máximo de caminhos independentes bem como o número de casos de testes necessários para exercitar todos os comandos pelo menos uma vez (PRESSMAN, 2016).

Pressman (2016) define como pode ser realizado o cálculo da Complexidade Ciclomática:

1. O número de regiões do GFC corresponde à complexidade ciclomática;
2. A complexidade ciclomática, V(G), para um GFC é definida como: V(G) = E - N + 2 Onde E é o número de arestas do grafo e N é o número de nós do grafo;
3. Complexidade Ciclomática, V(G), para um GFC é definida como: V(G) = P + 1 onde P é o número de nós predicativos contidos no grafo;

2.2 Meta-heurísticas

Uma meta-heurística é tida como uma estrutura algorítmica geral que pode ser posta a distintos problemas de otimização com modificações adaptativas para torná-las pertinentes a resolução de um problema específico (BLUM; ROLI, 2003).

Gendreau e Potvin (2010) definem uma meta-heurística como sendo um método de solução que organiza a interação entre procedimentos que buscam soluções ótimas ao mesmo tempo utilizam de estratégias capazes de escapar de ótimos locais, realizando uma busca robusta no espaço de solução.

Em resumo, meta-heurísticas possuem as seguintes propriedades (BLUM; ROLI, 2003):

1. Meta-heurísticas são estratégias que “guiam” o processo de busca;
2. O objetivo é explorar com eficiência o espaço de pesquisa para encontrar soluções ideais, ou quase ideais;
3. As técnicas que constituem algoritmos meta-heurísticos variam de procedimentos simples de busca local a processos complexos de aprendizado;
4. Os algoritmos meta-heurísticos são aproximados e geralmente não determinísticos;
5. Podem incorporar mecanismos para evitar ficar preso em áreas confinadas do espaço de busca;

6. Os conceitos básicos das meta-heurísticas permitem uma descrição abstrata do nível;
7. As meta-heurísticas não são específicas do problema.
8. As meta-heurísticas podem fazer uso de conhecimentos específicos de domínio na forma de heurísticas controladas pela estratégia de nível superior;
9. Atualmente, as meta-heurísticas mais avançadas usam a experiência de pesquisa (incorporada em alguma forma de memória) para orientar a pesquisa;

As meta-heurísticas em suma não são capazes de garantir a solução ótima, equiparando-se aos procedimentos exatos, que para problemas do mundo real (com alta complexidade) muitas vezes se mostraram incapazes de encontrar soluções ótima (GENDREAU; POTVIN, 2010). Além disso, Gendreau e Potvin (2010) menciona que as melhores implementações de métodos exatos incorporam estratégias metaheurísticas dentro deles.

Com tudo, pode-se dizer que as meta-heurísticas são estratégias usadas para investigar espaços de pesquisa compostas por diversificados métodos matemáticos (BLUM; ROLI, 2003).

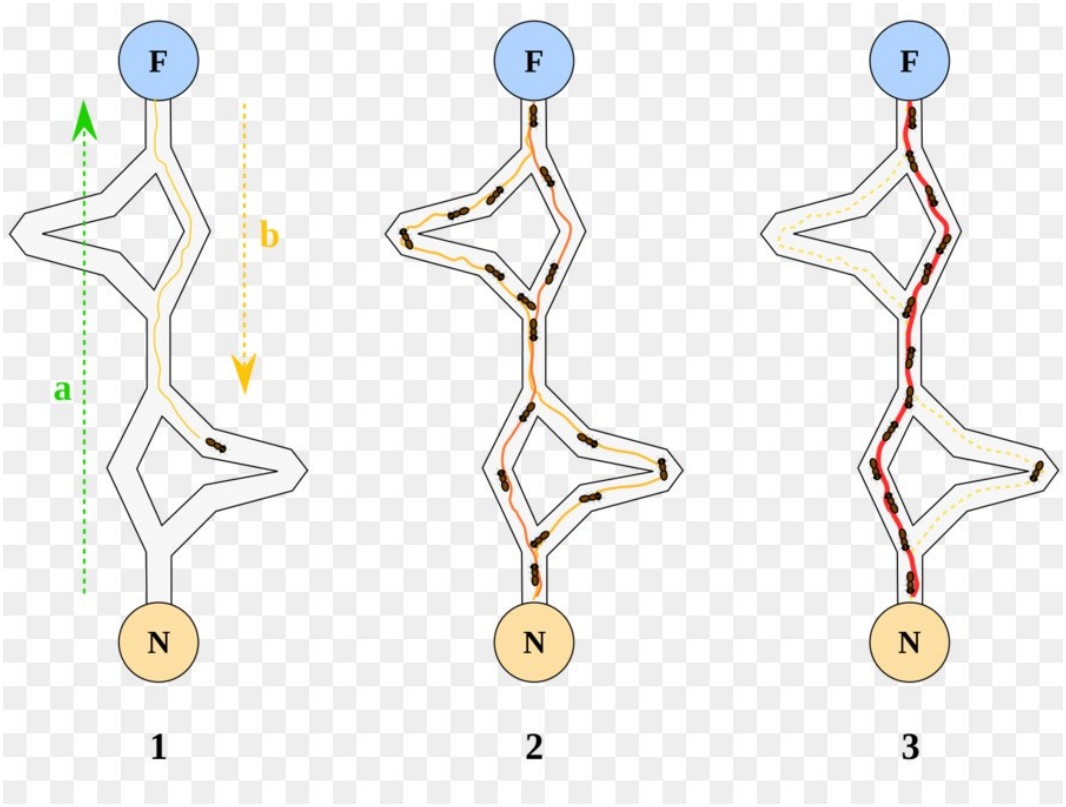
2.2.1 Meta-heurística: Otimização por Colônia de Formigas

O ACO, definida por Dorigo, Di Caro and Gambardella em 1999, é uma meta-heurística utilizada para resolver problemas computacionais de otimização (MONTES, 2010). Este algoritmo bio-inspirado é baseado no comportamento das formigas no processo de busca de alimento (RANJAN *et al.*, 2009).

Em seu comportamento natural, as formigas partem do formigueiro em busca de alimentos andando dispersas em várias direções e depositando feromônio no caminho percorrido. Ao encontrar a fonte de alimentos, retornam ao formigueiro pelo rastro de feromônio deixado, assim intensificando a quantidade de feromônio naquele caminho que pode ser percebido por outras formigas (LATTARO, 2017).

Ao sentir o feromônio depositado em um caminho, as formigas poderão escolher se seguem a rota percebida ou se irão continuar a busca por outra rota, sabendo que: quanto mais feromônio for sentido, existirá mais probabilidade da formiga seguir a rota percebida (FERREIRA, 2018). Como é ilustrada na Figura 3.

Figura 3 – Comportamento das Formigas



Fonte: Ferreira (2018).

O passo 1 da Figura 3, ilustra o inicio do processo na etapa em que a formiga parte do formigueiro em busca da fonte de comida. O passo 2, ilustra a etapa em que as demais formigas buscam pela fonte de comida de forma aleatória percorrendo outros caminhos. O passo 3, ilustra exatamente o momento onde todas as formigas intensificam o feromônio em um único caminho, sendo esse o principal caminho a ser percorrido.

Dorigo e Stutzle (2004) apresentam os conceitos base que serviram para a definição e a formulação do ACO, no qual o processo é definido como a interação entre os procedimentos seguintes e conforme é ilustrado no algoritmo 1:

- **Construir Soluções para Formigas:** as formigas percorrem de forma estocástica os estados da modelagem do problema e avaliam a solução (parcial) que será informada para o procedimento de Atualização de Feromônio;
- **Atualização de feromônio:** este processo modifica as trilhas aumentando a quantidade de feromônio ou diminuindo (evaporando) de acordo com a avaliação da formiga, fazendo com que seja gerado uma boa solução (com bastante feromônio) ou evitando uma convergência muito rápida do algoritmo, favorecendo, portanto, a exploração de novas áreas do espaço de busca.
- **Ações Daemon:** são ações que podem, por exemplo, serem ativadas para decidir se é útil ou não depositar feromônio adicional para polarizar o processo de pesquisa.

Algoritmo 1: Algoritmo de Otimização por Colônia de Formiga

```
Entrada: Entrada do Algoritmo
início
    Atribua os valores dos parâmetros;
    Inicialize as trilhas de feromônios;
    Enquanto não atingir o critério de parada para cada formiga faça
        Construa as Soluções;
        Atualize o Feromônio;
        Realize ações Daemon (opcional) ;
    fim
fim
```

O algoritmo 1 serviu como base para o desenvolvimento do algoritmo usado neste trabalho. Como complemento às Ações Daemon citadas acima, Lattaro (2017) explana que uma vertente para otimização da meta-heurística pode ser dada através do elitismo, assim, apenas a formiga que encontrou a melhor solução até o momento deposita o feromônio no caminho encontrado.

2.3 Teste de Software Baseado em Busca

Desde 1976, técnicas de otimização são utilizadas para automação de geração de dados de teste. A *Search-Based Software Testing (SBST)* é o termo dado para a automatização de alguma atividade de teste com o apoio do uso de meta-heurísticas (MCMINN, 2011).

As principais atividades de testes que vem sendo estudadas e automatizadas ao longo dos anos, são: teste funcionais, priorização de casos de testes, teste de integração, teste unitário, teste de regressão entre outras (MCMINN, 2011). Sahoo e Ray (2018) fazem uma revisão sistemática das principais técnicas de pesquisa utilizadas para automatização de algumas destas tarefas, listando: *Genetic Algorithm (GA)*, *Simulated Annealing (SA)*, *Artificial Bee Colony (ABC)*, *Ant Colony Optimization (ACO)*.

No trabalho de Harman *et al.* (2015), é relatado a necessidade de mais pesquisas para os desafios e estratégias relacionados à SBST. A estratégia *Search-Based Test Strategy Identification (SBTSI)* visa automatizar e otimizar a priorização de testes para o teste de regressão; *Search-Based Energy Testing (SBET)* estratégia de automatização e otimização de testes não-funcionais de consumo de energia; *Multi-objective Search Based Testing (MoSBaT)* automatização e otimização de testes multi-objetivo, como ponto chave para questões do meio das industrias de software.

3 TRABALHOS RELACIONADOS

Nesta seção, serão apresentados os trabalhos relacionados nos quais esta pesquisa se baseia. Os trabalhos foram pesquisados utilizando como base as palavras-chaves: *Test Case Generation*, *Metaheurístics*, *Ant Colony Optimization*, *Search-based Software Testing*. Os trabalhos foram selecionados com base no critério de Abordagem Utilizada, com ênfase nos trabalhos desenvolvidos na diretriz de Teste de ramificação.

Em seu trabalho, Arifiani e Rochimah (2016) propõem a geração de casos de teste com uma abordagem que utiliza Diagrama de Maquina de Estados UML em união com Testes de Caixa Cinza. A proposta do trabalho é comparar a geração de casos de testes desta abordagem com abordagens baseadas em Testes Estruturais feitas em pesquisas prévias. É justificado o uso do Diagrama de Estados como sendo a melhor forma de modelar um código fonte por conta da sua capacidade de cobrir toda a estrutura melhor que outros diagramas UML.

Seguindo a metodologia proposta, para um determinado código fonte é gerado um diagrama de estados com a técnica de Caixa Cinza, onde, para cada caso de teste gerado, será analisado e comparado a capacidade de cobertura de um ramo com casos de teste gerados puramente com Teste Estrutural. O diagrama de estados é validado manualmente, e após a validação, é gerado um grafo que será usado no ACO. Para cada entrada de um programa teste (programa que será testado) sera gerado um valor que pertencerá ao domínio do programa, e cada formiga conterá esses valores de entradas. Ao percorrer as ramificações, cada entrada vai ser analisada com base em critérios de cobertura, e ao final será escolhido os valores que tiveram a melhor avaliação.

Em conclusão, os autores justificam que o diagrama de máquina de estados UML possui a mesma estrutura que o grafo gerado a partir do código-fonte, mas que os resultados da abordagem com Caixa Cinza foram melhores do que a abordagem com Caixa Branca.

No trabalho de Sharifipour *et al.* (2017), é proposta uma variação memética (evolutiva) do ACO utilizado para geração de dados de teste. Os autores introduzem uma nova abordagem para a funcionalidade de feromônio, característico dessa meta-heurística, que as formigas tendem a não escolher caminhos mais cobertos do programa, assim reforçando a exploração da pesquisa por novos.

Sharifipour *et al.* (2017) aborda o uso de duas funções de aptidão: uma função booleana que maximiza a cobertura do ramo, caso este possua ao menos uma ramificação ainda não coberta; a segunda função é exercitada de acordo com os valores retornados da primeira função, onde só é exercitada caso o retorno da função 1 for diferente de Verdadeiro.

Para avaliar o desempenho do algoritmo desenvolvido, Sharifipour *et al.* (2017) realiza uma comparação com o algoritmo básico desenvolvido por Mao et al. São estão propostas três implementações do ACO memético, seguindo as extensões:

1. Incorporating (1 + 1) -ES para movimentos locais de formigas ao invés de persegui-los aleatoriamente.;
2. Apresentando nova definição de trilhas de feromônio e incluindo informações sobre feromônios para função de aptidão;
3. Definir da segunda função de aptidão booleana adicional para a avaliação de soluções candidatas;

Os resultados dos experimentos revelam que as duas ultimas extensões melhoraram significativamente a cobertura do ramo onde, em alguns casos, o ACO memético alcançara a cobertura total.

Mao *et al.* (2014) traz em seu trabalho a proposta de adaptar o ACO na geração de dados de testes para testes estruturais. Como motivação de que o ACO não foi profundamente estudado como uma abordagem de geração de casos de teste, eles realizam uma reforma no ACO Básico para realizar o comportamento de pesquisa nos intervalos das variáveis de entrada do programa, também realizam interações rastreando informações para direcionar a otimização e, além disso, desenvolvem uma função *fitness* que considera o nível de aninhamento do ramo e o tipo de predicado para cobertura do ramo.

Como principais contribuições, Mao *et al.* (2014) apresenta o seguintes tópicos:

- O ACO básico é modificado para atender ao problema de geração de dados de teste estrutural englobando também estratégias como transferência local, transferência global e atualização de feromônio;
- Comparativo entre os algoritmos *Simulated Annealing*, *Genetic Algorithm* e *Particle Swarm Optimization* adaptados ao mesmo problema, onde o algoritmo proposto no trabalho tem melhor desempenho que o *Simulated Annealing* e *Genetic Algorithm*, também sendo comparável ao *Particle Swarm Optimization*;
- A analise de parâmetros é conduzida no trabalho para fornecer diretrizes da configuração do ACO no problema de geração de dados de teste;

Ao final de seu estudo, Mao *et al.* (2014) apresenta os resultados colhidos e os compara em três segmentos: análise estática, análise de estabilidade e análise de sensibilidade de parâmetro. Para análise estática é apresentado, de modo geral, o desempenho dos algoritmos mencionados acima de acordo com as métricas de Média de Cobertura, Taxa de Sucesso, Média de geração e Média de Tempo. Para análise de Estabilidade, é realizado um experimento de 1000 repetições, nas quais em apenas 10 o ACO falhou, quando verificado em tempo de cobertura. Apesar dos bons resultados com os segmentos anteriores, para a sensibilidade de parâmetro o ACO não obteve uma eficiência agradável para os tipos de dados *String* ou *Object*, cujo o desenvolvimento do algoritmo para análises de dados complexos é também uma tarefa interessante e valiosa.

3.0.1 Resumo dos trabalhos

O Quadro 1 mostra uma breve comparação entre a abordagem proposta nesta pesquisa com as abordagens dos trabalhos relacionados, mostrando os objetivos de cada e suas respectivas técnicas utilizadas para geração de dados testes automática.

Quadro 1 – Comparação de trabalhos

Trabalho	Objetivos	Técnica utilizada
Mao *et al.* (2014)	Realizar a geração de dados de teste com uma implementação básica do ACO e comparar com o desempenho de outras meta-heurísticas implementadas.	ACO + Árvore de sintaxe abstrata
Arifiani e Rochimah (2016)	Realizar a geração de dados de testes através da modelagem de um Diagrama de Máquina de Estado UML implementando o conceito de caixa cinza e comparação com abordagens puramente de caixa branca.	ACO + Diagrama de Máquina de Estados UML
Sharifipour *et al.* (2017)	Realizar a geração de dados de teste com uma versão memética do ACO e compará-la com os resultados de Mao *et al.* (2014).	ACO + Árvore de sintaxe abstrata
Este trabalho	Realizar a geração de dados de testes através da modelagem de um GFC, implementando o ACO básico e focando na busca de caminhos independentes que atinjam boas porcentagens de cobertura com base na complexidade ciclomática.	ACO + Grafo de Fluxo de Controle

Fonte: O autor (2021).

Por fim, foi possível observar as diferentes formas como estes trabalhos conduzem a geração de dados de teste. Nestes trabalhos, também foi observado que nenhum conduz sua abordagem baseada na cobertura de caminho independente, sendo este um ponto considerável que foi abordado neste trabalho.

4 ALGORITMO DE MODELAGEM E GERAÇÃO DE CAMINHOS INDEPENDENTES DE UM GFC

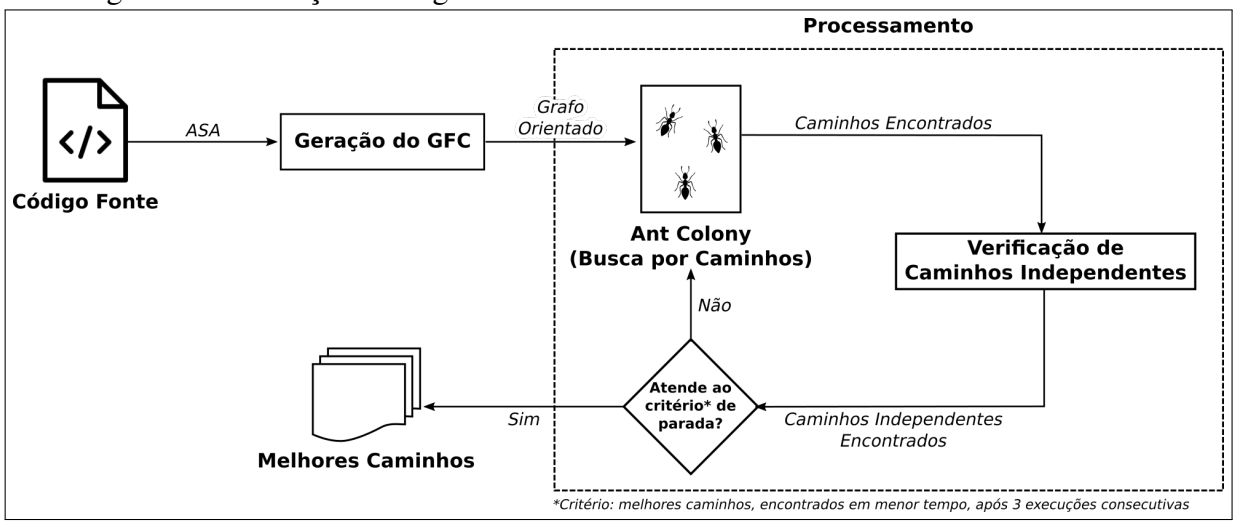
Neste capítulo serão descritas as etapas de desenvolvimento e aplicação do algoritmo proposto para este trabalho. Serão mencionadas as técnicas adotadas e as tecnologias utilizadas para realizar a adaptação da meta-heurística ACO com o propósito de encontrar caminhos independentes em um GFC. Assim, fomentando a geração automática de dados de testes com caminhos distintos e potencializando o alcance da qualidade esperada pela funcionalidade testada com a cobertura dos caminhos encontrados.

Como passo inicial, foram estudas as técnicas e abordagens empregadas para modelar o código escrito na linguagem *Python* em um grafo. Após o estudo, foram identificadas e definidas as seguintes etapas para realizar esta modelagem: ler o arquivo que contém o código da função, obter a Árvore de Sintaxe Abstrata (ASA) desta função e, por último, gerar um grafo com base na ASA. Para isto, utilizou-se os métodos disponibilizados pela biblioteca *StatiCFG* (LEONARD, 2019) que contém todo este mecanismo preparado para uso.

Com um arquivo contendo o código da função à ser testada, o processamento inicia com os passos: ler a ASA deste código, identificar todos os blocos de comandos, separando-os em nós e, também, definir suas adjacências em arestas, assim representando a sequência dos fluxos. Ao final do processamento, é retornado um arquivo com a extensão ".dot" representando o grafo não orientado do código passado. Com o arquivo ".dot" gerado, é executado o mecanismo de leitura e conversão do "grafo.dot" em um grafo orientado, representando assim o GFC. Toda esta parte de conversão do grafo não orientado em grafo orientado foi desenvolvido com o métodos disponibilizados pela biblioteca *NetworkX* (NETWORKX, 2014), que é amplamente utilizada para modelar diversos tipos de grafos. Este e os demais passos são ilustrados pela Figura 4 (o código completo do algoritmo proposto está disponível no gitHub do autor[1]).

[1] https://github.com/samuelcostab/testDataGeneration

Figura 4 – Ilustração do Algoritmo



Fonte: O autor (2021)

Após a etapa de modelagem do GFC, o algoritmo desenvolvido segue para o processamento de busca dos caminhos independentes e escolha dos caminhos que melhor atendem ao critério de cobertura. Nesta etapa foram definidos alguns dos principais parâmetros utilizados no desenvolvimento, como: a Complexidade Ciclomática (CC) que define a quantidade base de caminhos independentes no GFC, melhor tempo de iteração, consecutividade dos caminhos dentre outros.

O algoritmo 2 mostra o pseudo-código de como é realizado o processo de busca dos caminhos. A estratégia utilizada, foi uma estratégia gulosa com escolha randômica para os nós adjacentes (vizinhos) ao nó verificado em cada iteração. A escolha aleatória é feita com a verificação de um valor (sendo 0 ou 1) que define, a cada iteração, como será feito a escolha e/ou ordenação dos nós adjacentes.

Na busca dos caminhos independentes, são realizadas diversas iterações de busca da meta-heurística para obter os caminhos do GFC e verificar se os mesmos são independentes, utilizando a função “Verifica Caminhos” exemplificada pelo algoritmo 3. Para cada caminho encontrado, a função citada analisa se o caminho tem pelo menos 1 nó "não conhecido"(ainda não visitado) considerado-o como um caminho independente, caso exista algum nó do grafo não coberto pelos caminhos encontrados então a busca é realizada novamente, caso contrário, o algoritmo segue para a atualização dos parâmetros salvando os novos valores de tempo de iteração, consecutividade e caminhos independentes encontrados.

Algoritmo 2: Busca de Caminhos

```
Entrada: G = Grafo, LNV = Lista de Nós Visitados
begin
    UNV = Último nó de LNV;
    LV = Lista de possíveis vizinhos;
    for cada vizinho(V) de UNV do
        if V não tiver sido visitado then
            Custo = feromônio presente no caminho de UNV até o V;
            LV = (Custo, V);
    end
    if LV tiver elementos then
        ES = valor aleatório: 0 ou 1;
        if ES = 1 then
            Seleciona um elemento de LV randomicamente;
            E atribui ao final de LNV;
        else
            Ordena LV por nível de feromônio;
            LNV atribui os nós de LV;
        end
    else
        Completa a busca atual;
        E retorna o caminho encontrado na iteração;
    end
end
```

Depois da atualização dos parâmetros, é verificado o critério de parada do processamento, que se dá por encontrar os melhores caminhos (com melhor tempo de iteração) três vezes consecutivas, assim retornando os caminhos independentes encontrados que serão utilizados como dados de teste.

Algoritmo 3: Verifica Caminhos

Entrada: P = Caminhos Encontrados, NVC = Novos Caminhos, G = Grafo

```
begin
    for cada caminho(C) de NVC do
        for cada nó(N) de C do
            if N já foi coberto por P then
                P incrementa a cobertura de N;
        end
    end
    if P cobre todos os nós de G then
        retorna os caminhos até aqui encontrados
    else
        Continua a Busca de Caminhos
    end
end
```

5 RESULTADOS E DISCUSSÃO

Neste capítulo serão apresentados os resultados obtidos com o desenvolvimento deste trabalho.

5.1 Geração do Grafo de Fluxo de Controle

A geração de um GFC tinha o intuito de modelar uma função escrita na linguagem *Python* a qual permitiria conduzir o ensaio da meta-heurística, fazendo com que a mesma executasse o processamento de otimização neste grafo.

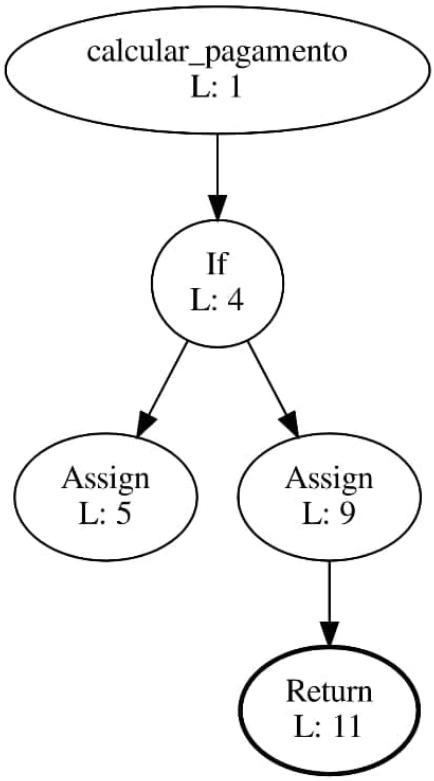
Inicialmente foi utilizada a biblioteca *PySoCa* (JEAN; VANIA, 2019), na qual foi possível gerar um GFC para a função "Calcular Pagamento" escrita na linguagem *Python*, ilustrada no Código-fonte 1. Esta função foi utilizada nos testes iniciais, já que a mesma contém apenas dois fluxos (complexidade baixa). Nesta função, é possível observar que cada linha que introduz algum novo bloco de comandos é representada por um nó do GFC, assim os nós representam as linhas: 1, 4, 5, 9 e 11, como mostra a Figura 5.

Código-fonte 1 – Função Calcula Pagamento.

```
1  def calcular_pagamento(qtd_horas, valor_hora):
2    horas = float(qtd_horas)
3    taxa = float(valor_hora)
4    if horas <= 40:
5      salario=horas*taxa
6
7
8    else:
9      h_excd = horas - 40
10     salario = 40*taxa+(h_excd*(1.5*taxa))
11   return salario
```

Porém, para o fluxo entre os comandos 5 e 11, foi identificado que não existe a aresta ligando os dois pontos. Na Figura 5 é possível observar que no bloco L5 (que representa as instruções a partir da linha 5 do arquivo) não possui ligação com o bloco L11, evidenciando um erro na modelagem do GFC.

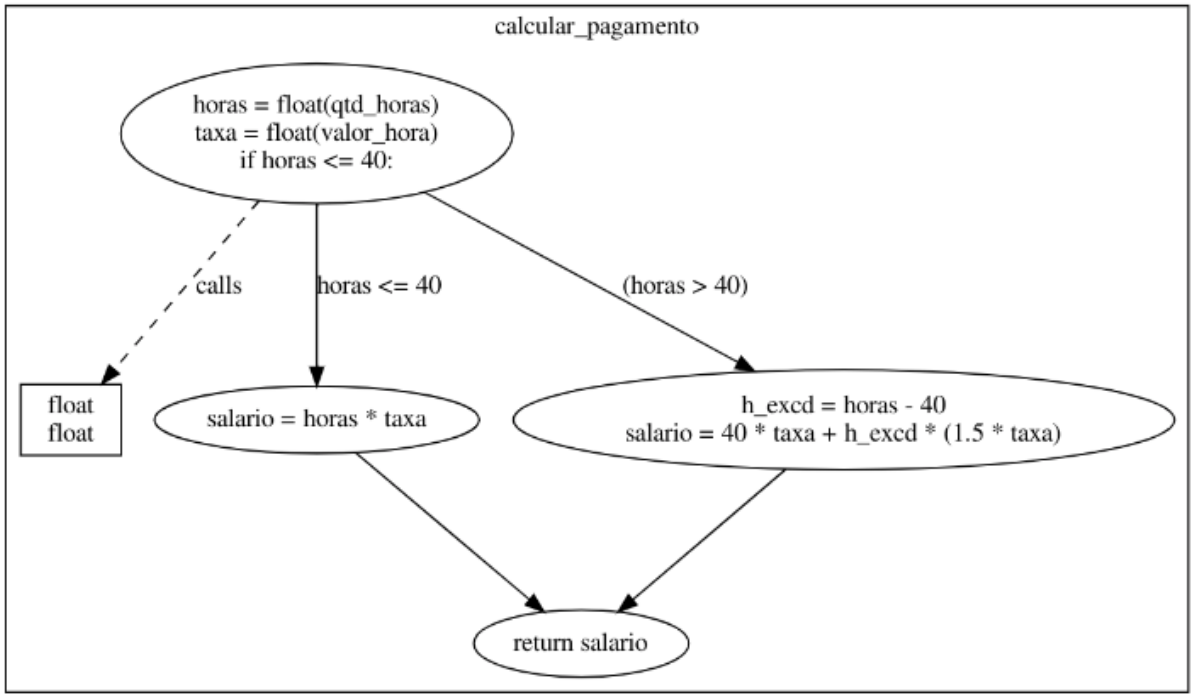
Figura 5 – GFC Modelado com PySoCa



Fonte: O autor (2021).

Para contornar este problema, foi então estudada a biblioteca *StatiCFG* (LEONARD, 2019) que possui, também, a finalidade de modelar o GFC. Então, após a *StatiCFG* ser adaptada e incorporada ao projeto, conseguiu-se obter o resultado esperado, sendo possível explorar os fluxos corretamente. A Figura 6 ilustra a modelagem esperada para essa função, ligando corretamente o que seria o bloco L5 ao bloco L11, como mencionado anteriormente. Como complemento, o GFC gerado pela biblioteca *StatiCFG* é mais rico em detalhes, contendo as informações dos blocos da função. E não somente o número da linha, quando comparado com o GFC da Figura 5 modelado pela biblioteca *PySoCa*.

Figura 6 – GFC Modelado com StaticCFG

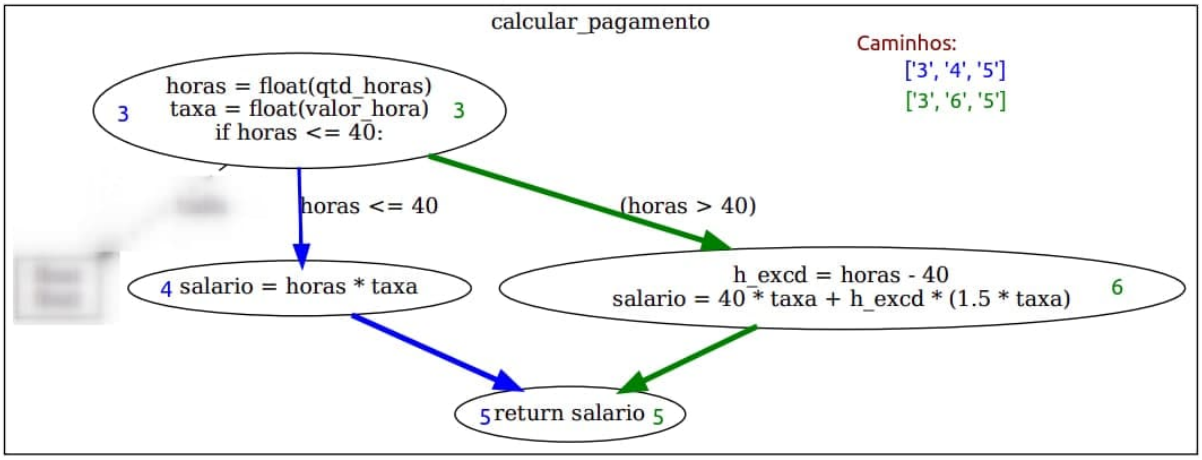


Fonte: O autor (2021).

5.2 Busca de Caminhos Independentes do GFC

Após gerar corretamente o GFC, foi então realizada a execução da meta-heurística para o processo de busca dos caminhos. O GFC foi modelado como um grafo orientado no qual foi utilizado como espaço de busca da meta-heurística. No processamento, a meta-heurística busca a diversificação dos caminhos percorridos pelas formiga e, como critério de parada, espera a obtenção dos caminhos quando estes possuíam os melhores tempos de retorno três vezes consecutivas. Após a finalização do ensaio, foram encontrados todos os caminhos do grafo de fluxo de controle estudado ilustrados na Figura 7.

Figura 7 – Ilustração dos Caminhos encontrados

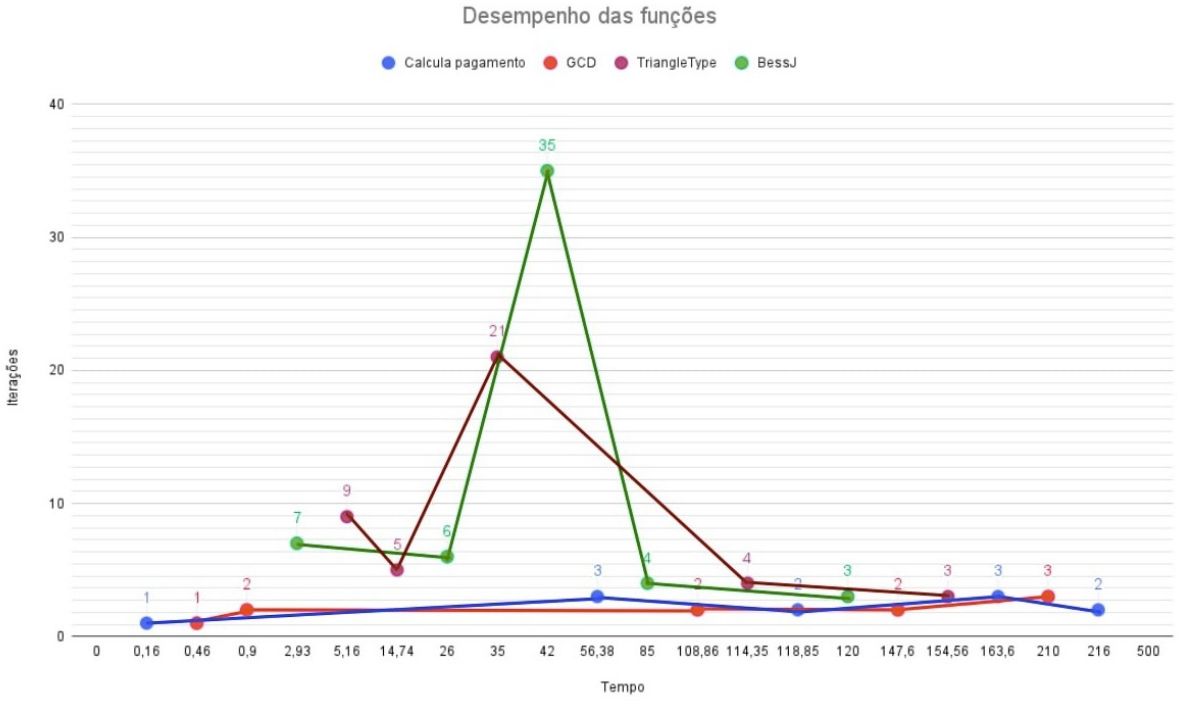


Fonte: O autor (2021).

Nesta etapa, também foi validado a capacidade de busca pelos caminhos independentes do algoritmo proposto com outras funções escritas em *Python*. Para estas, foi possível gerar o GFC e encontrar caminhos independentes que servissem como dados de testes. Durante a busca dos caminhos foi analisado o desempenho do algoritmo relacionado ao custo de iterações (ilustrado nas Figura 8 e Quadro 2) focando na capacidade de minimizá-lo (menor quantidade de iterações possível) e com base nos critérios definidos pela abordagem deste trabalho. Todos os resultados também estão disponíveis no gitHub do autor[2].

[2] https://github.com/samuelcostab/testDataGeneration/tree/experimentos

Figura 8 – Desempenho das Funções



Fonte: O autor (2021).

Quadro 2 – Desempenho das Funções - Valores

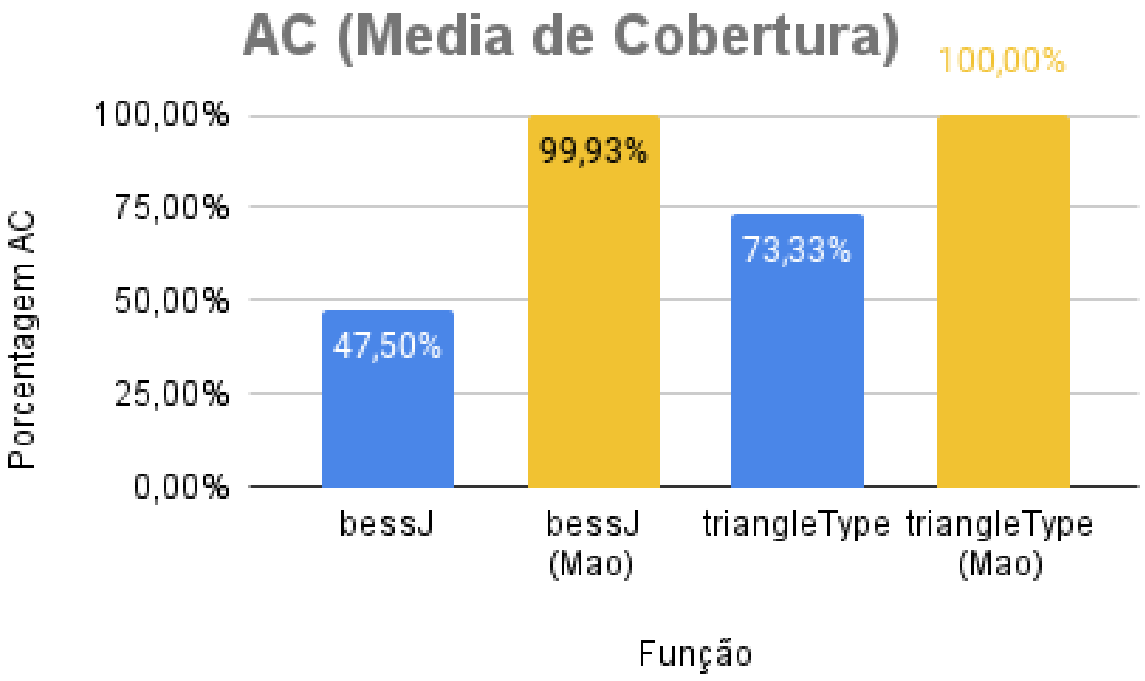
Função	Cobertura	Iterações	Melhor Iteração	Tempo de Execução (seg)
BessJ	14	7	7	2.93
BessJ	23	6	6	26
BessJ	28	35	6	42
BessJ	17	4	4	85
BessJ	19	3	3	120
TriangleType	31	9	9	5.16
TriangleType	26	5	5	14.74
TriangleType	43	21	5	35
TriangleType	23	4	4	114.35
TriangleType	20	3	3	154.56
GCD	4	1	1	0.46
GCD	7	2	1	0.9
GCD	6	2	1	108.86
GCD	5	2	1	147.6
GCD	7	3	1	210
Calc. Pagamento	3	1	1	0.16
Calc. Pagamento	3	3	1	56.38
Calc. Pagamento	3	2	1	118.85
Calc. Pagamento	3	3	1	163.6
Calc. Pagamento	3	2	1	216

Fonte: O autor (2021).

Foi realizado um estudo comparativo com a geração de caminhos para duas funções utilizadas por Mao *et al.* (2014). As funções foram escolhidas de acordo com a complexidade das mesmas, onde os autores definem consecutivamente: *triangleType* (complexidade baixa) e *bessJ* (complexidade alta) (SOFTWARETESTING, 2005). Para este estudo comparativo, as condições de parada da função e os parâmetros da meta-heurística foram adaptadas de acordo com os parâmetros usados por Mao *et al.* (2014).

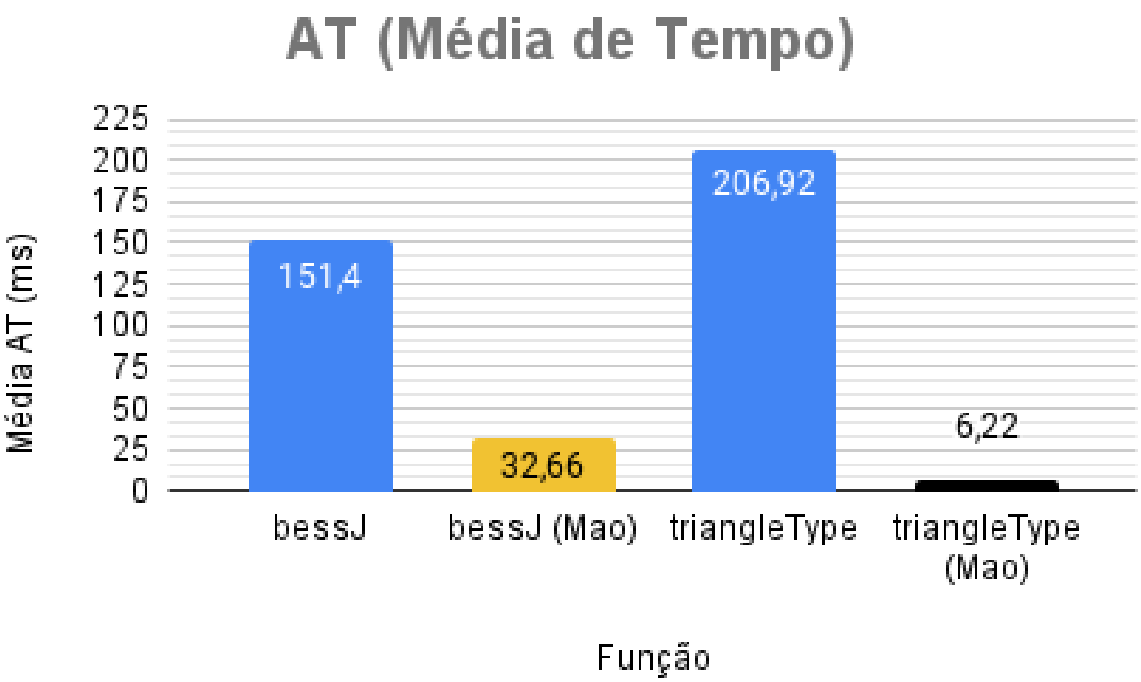
Apesar de conseguir retornar os caminhos independentes que testassem a função, de acordo com os parâmetros usados no algoritmo proposto, os resultados foram inferiores aos resultados de Mao *et al.* (2014). Nas figuras 9, 10 e 11 são ilustrados os desempenhos de ambos os algoritmos com relação às métricas de Média de Cobertura (AC), Média de Tempo (AT) e Taxa de Sucesso(SR).

Figura 9 – Gráfico Média de Cobertura



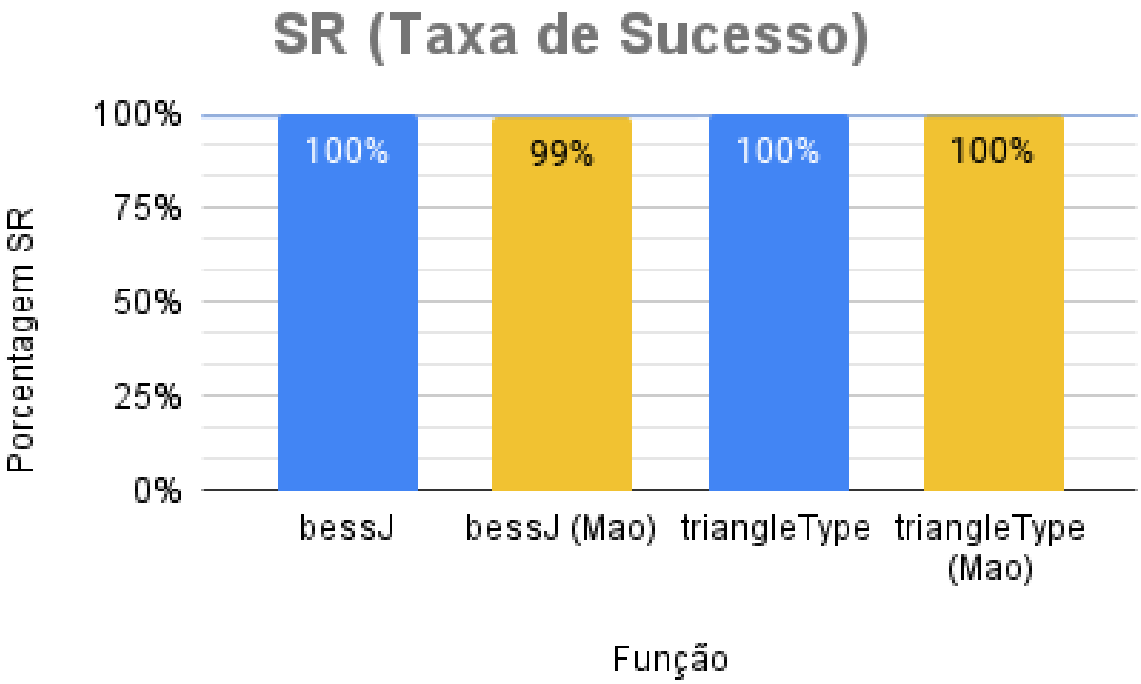
Fonte: O autor (2021).

Figura 10 – Gráfico Média de Tempo



Fonte: O autor (2021).

Figura 11 – Gráfico Taxa de Sucesso



Fonte: O autor (2021).

5.3 Análise de Resultados

O algoritmo proposto por este trabalho se mostrou capaz de encontrar caminhos independentes em códigos escritos em *Python*, apesar de um desempenho inferior comparado-o com outros algoritmos, à exemplo o algoritmo de Mao *et al.* (2014).

Foi possível analisar que a minimização de iterações pode não ser uma abordagem ideal para este algoritmo. De acordo com os resultados da **??** é possível analisar que a quantidade de caminhos independentes é superior nos resultados que tiveram maior número iterações do que nos resultados de menor número de iterações. Assim, uma possível melhoria para este algoritmo seria uma abordagem voltada para a maximização de iterações, buscando sempre os valores com maior número iterações.

Para algumas das funções mencionadas acima conseguiu-se encontrar os caminhos que cobrissem 100% o GFC em um tempo inferior à 1 segundo. Já para as funções mais complexas (que possuem mais condições e fluxos), a geração de caminhos com 100% de cobertura levou um tempo considerável ultrapassando 5 minutos. Com tudo, foi analisado que para funções mais simples o algoritmo consegue encontrar próximo do 100% de cobertura e para algoritmos mais complexos, o algoritmo consegue encontrar uma boa quantidade de caminhos mas não chega a garantir 100% de cobertura.

Na figura 8 é possível observar o ponto máximo de iterações atingido pelo algoritmo deste trabalho para as funções analisadas, evidenciando que em alguns casos o algoritmo fica processando muito mais possibilidades. Na maioria dos casos foi possível encontrar mais caminhos nesse pico de iterações, mas também foi possível encontrar boa quantidade de caminhos em quantidades de iterações mais baixas, porém com recorrência menor.

Como a análise de cobertura pelo algoritmo desenvolvido é com base na complexidade ciclomática, calculada basicamente pela quantidade de nós e arestas do grafo, para as funções de complexidade mais básicas foi possível encontrar e percorrer todos caminhos existentes no GFC desta função. Fato também relacionado ao processo de busca e escolha da meta-heurística desenvolvida neste trabalho, que é mais simples do que comparado-a com as implementações de outros autores.

Para os casos em que as funções são mais complexas e com maiores combinações de caminhos, a definições de parâmetros e processamentos existentes no algoritmo retornam uma quantidade de caminhos independentes não suficientes para cobrir 100% das possibilidades existentes no GFC da função analisada.

6 CONCLUSÕES E TRABALHOS FUTUROS

A partir dos estudos desenvolvidos neste trabalho foi possível atingir os objetivos especificados, assim realizando a geração de um GFC a partir de uma função escrita na linguagem *Python*, sendo este, usado para a obtenção dos caminhos independentes fomentando a geração automática de dados de testes estáticos.

Com este trabalho foi possível observar o quão desafiador é a geração automática de dados de testes, uma vez que alguns fatores são muito relevantes para esta área: tempo de geração, cobertura, otimização, entre outros. Apesar disso, este trabalho possibilita uma abordagem que pode ser melhorada.

Algumas melhorias podem ser desenvolvidas para garantir mais qualidade e melhor desempenho do algoritmo. Uma primeira melhoria seria realizar a adaptação da função de maximização das iterações feitas pelo algoritmo deste trabalho, assim realizando o experimento com geração de caminhos buscando o maior número de iterações (mais caminhos retornados) e realizando a comparação direta com estes resultados. Outra melhoria, seria relacionado com os métodos de buscas da meta-heurística tornando-a mais rápida e assim obtendo ganhos em termos de performance. Em complemento, desenvolver um interpretador que a partir do "arquivo.dot" juntamente com os caminhos retornados por este trabalho gerasse os valores atribuídos às variáveis de forma automática, assim deixando o processo ainda mais automático.

Em relação à trabalhos futuros, pode-se citar a geração automática de dados complexos, como *Object* ou *String*, que ainda é um desafio em aberto e precisa ser melhor estudado, assim fomentaria em grande escala a geração de dados de teste. Outra questão que também pode ser abordada desrespeito a priorização de CT's, que a partir da modelagem pode-se realizar uma priorização dos caminhos à serem testados, assim deixando o testes mais rápidos e mais assertivos. Também seria de grande valor uma abordagem de geração automática de testes desenvolvida com integração contínua, podendo gerar dados de teste em uma esteira e assim auxiliar os analistas com a verificação das funções mais rapidamente.

Com tudo, foi possível contribuir para os estudos da área de testes apresentando uma abordagem de automatização para área de testes unitários e estáticos, elaborando uma estratégia de desenvolvimento do algoritmo no qual pode ser mantido e evoluído por trabalhos futuros e que pode auxiliar no desenvolvimento de aplicações e sistemas. Além disso, comentar alguns dos desafios encontrados e estudados no campo da *Seach Based Software Testing*.

REFERÊNCIAS

ALVES, W. Custo da qualidade e seus efeitos na ambidestria organizacional e no desempenho da industria de software. REPOSITÓRIO INSTITUCIONAL DA UNIVERSIDADE ESTADUAL DE MARINGÁ, 2020. Disponível em: <http://repositorio.uem.br:8080/jspui/bitstream/1/6035/1/Welliton\%20Felipe\%20Alves\%20Miranda_2020.pdf>.

ARIFIANI, S.; ROCHIMAH, S. Generating test data using ant colony optimization (aco) algorithm and uml state machine diagram in gray box testing approach. International Seminar on Application for Technology of Information and Communication, 2016.

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys, Vol. 35, No. 3, September 2003, pp. 268–308., 2003.

CHIAVEGATTO, S.; VIEIRA; MALVEZZI. Desenvolvimento orientado a comportamento com testes automatizados utilizando jbehave e selenium. Anais do Encontro Regional de ComputaÄÅo e Sistemas de InformaÄÅo, 2013.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: IBSN 8535283528, 2016. v. 2 edição.

DIAS, L. **Os Benefícios do Teste Automatizado**. 2018. Disponível em: <https://medium.com/test-up/os-benef\%C3\%ADcios-do-teste-automatizado-88c3cd9bf3aa>. Acesso em: 30 jun. 2021.

DORIGO, M.; STUTZLE, T. **Ant Colony Optimization**. [S.l.]: Massachusetts Institute of Technology, 2004. v. 1 Edition.

FERREIRA, F. **Formigas podem ajudar o Caixeiro Viajante a ganhar um milhão de dólares**. 2018. Disponível em: <https://www.deviante.com.br/noticias/ciencia/formigas-podem-ajudar-o-caixeiro-viajante-a-ganhar-um-milhao-de-dolares>. Acesso em: 16 fev. 2021.

GENDREAU, M.; POTVIN, J.-Y. **Handbook of Metaheuristics**. [S.l.]: International Series in Operations Research Management Science, 2010. v. 2 Edition.

HARMAN, M.; JIA, Y.; ZHAN, Y. Achievements, open problems and challenges for search based software testing. 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015.

JEAN, P.; VANIA, N. **PySoCa: uma ferramenta para teste estrutural de programas escritos em Python**. 2019. Disponível em: <https://github.com/vaniatestingtools/pysoca>.

J.SILVA; G.VIANA; MACHADO, G.; SILVA, R. O processo de teste de software. Tecnologias em Projeção, v. 7, n. 2, 2016.

LATTARO, A. **Entendendo o Ant Colony Optimization**. 2017. Disponível em: <https://imasters.com.br/desenvolvimento/entendendo-o-ant-colony-optimization>. Acesso em: 16 nov. 2019.

LEONARD, T. **StatiCFG: Python3 control flow graph generator**. 2019. Disponível em: <https://github.com/coetaur0/staticfg>.

LEWIS, W. E. **Software Testing and Continuous Quality Improvement**. [S.l.]: Auerbach Publications, 2009. v. 3 Edition.

MAO, C.; LICHUAN, X.; XINXIN, Y.; JINFU, C. Aadapting ant colony optimization to generate test data for software structural testing. Swarm and Evolutionary Computation, 2014.

MARTINHO, F. **Qualidade, Qualidade de Software e Garantia da Qualidade de Software são as mesmas coisas**. 2021. Disponível em: <http://www.linhadecodigo.com.br/artigo/1712/qualidade-qualidade-de-software-e-garantia-da-qualidade-de-software-sao-as-mesmas-coisas>. Acesso em: 27 jun. 2021.

MCMINN, P. Search-based software testing: Past, present and future. In: **International Workshop on Search-Based Software Testing (SBST 2011)**. [S.l.]: IEEE, 2011. p. 153–163.

MONTES, M. A. **About Ant Colony Optimization**. 2010. Disponível em: <http://www.aco-metaheuristic.org/about.html>. Acesso em: 22 fev. 2020.

MYERS, G. J.; BADGETT, T.; SANDLER, C. **THE ART OF SOFTWARE TESTING**. [S.l.]: John Wiley Sons, Inc, 2012. v. 3 Edition.

NETWORKX. **NetworkX: Analysis in Python**. 2014. Disponível em: <https://networkx.org>.

PRESSMAN, R. S. **Engenharia de software: uma abordagem profissional**. [S.l.]: AMGH Editora Ltda, 2016. v. 8.

RANJAN; BAB, K.; RAGHURAMA, G. An approach of optimal path generation using ant colony optimization. TENCON 2009, 2009.

ROSADO, R. Comparativo entre o gerenciamento tradicional e o ágil de projetos de desenvolvimento de software. Repositório Anima Educação, 2020. Disponível em: <https://repositorio.animaeducacao.com.br/bitstream/ANIMA/3718/1/Artigo_Renan_apos_defesa.pdf>.

SAHOO, R. R.; RAY, M. Metaheuristic techniques for test case generation:a review. Journal of Information Technology Research March/2018, 2018.

SHARIFIPOUR, H.; SHAKERI, M.; HAGHIGHI, H. Structural test data generation using a memetic ant colony optimization based on evolution strategies. Swarm and Evolutionary Computation, 2017.

SOFTWARETESTING. **Software Testing**. 2005. Disponível em: <https://tracer.lcc.uma.es/problems/testing/index.html>. Acesso em: 13 Mai. 2021.

SOUZA; GASPAROTTO. A importÂncia da atividade de teste no desenvolvimento de software. XXXIII ENCONTRO NACIONAL DE ENGENHARIA DE PRODUCAO, 2013.

SUMMERVILLE, I. **Introdução ao teste de software**. [S.l.]: Peason Education do Brasil, 2011. v. 9.

VASCO, T. V. Teste de funções por cobertura do grafo de fluxo de controle. Faculdade de Ciências da Universidade de Lisboa, 2016.

APÊNDICE A – CÓDIGOS-FONTES UTILIZADOS NA PESQUISA

Código-fonte 2 – BessJ

```
def bessj(n, x):
    j, jsum, m = 0
    ax, bj, bjm, bjp, sum, tox, ans = 0.0

    tmp = n

    if (n < 0):
        nrerror("Index n less than 0 in bessj")
    elif (n == 0):
        return bessj0(x)
    elif (n == 1):
        return bessj1(x)

    ax = fabs(x)
    if (ax == 0.0):
        return 0.0
    elif (ax > tmp):
        tox = 2.0/ax
        bjm = bessj0(ax)
        bj = bessj1(ax)
        for j in range(n):
            bjp = j*tox*bj-bjm
            bjm = bj
            bj = bjp
        ans = bj
    else:
        tox = 2.0/ax
        m = 2*((n+sqrt(ACC*n))/2)
        jsum = 0
```

```
30          bjp = ans = sum = 0.0
31          bj = 1.0
32          j=m
33          for j in range(0):
34              bjm = j*tox*bj-bjp
35              bjp = bj
36              bj = bjm
37              tmp = fabs(bj)
38              if (tmp > BIGNO):
39                  bj *= BIGNI
40                  bjp *= BIGNI
41                  ans *= BIGNI
42                  sum *= BIGNI
43              if (jsum):
44                  sum+= bj
45              jsum = not jsum
46              if (j == n):
47                  ans = bjp
48
49          sum = 2.0*sum-bj
50          ans /= sum
51
52      if (x < 0.0 & (n & 1)):
53          return -ans
54      else:
55          return ans
```

Código-fonte 3 – TriangleType

```
def triang(i, j, k):
    tri = 0
    if ((i <= 0) | (j <= 0) | (k <= 0)):
        return 4

    tri = 0
    if (i == j):
        tri += 1
    if (i == k):
        tri += 2
    if (j == k):
        tri += 3
    if (tri == 0):
        if ((i+j <= k) | (j+k <= i) | (i+k <= j)):
            tri = 4
        else:
            tri = 1
        return tri

    if (tri > 3):
        tri = 3
    elif ((tri == 1) & (i+j > k)):
        tri = 2
    elif ((tri == 2) & (i+k > j)):
        tri = 2
    elif ((tri == 3) & (j+k > i)):
        tri = 2
    else:
        tri = 4
    return tri
```

Código-fonte 4 – GCD

```
def gcd():
    a, b, tmp

    a = value1
    b = value2

    if (a < 0):
      a = -a

    if (b < 0):
        b = -b

    if (a == 0 | b == 0):
        print("The input values must be greater than zero\n
           ")
        return

    while (b > 0):
        tmp = a % b
        a = b
        b = tmp

    print("Result: %i\n", a)
```