UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

DEUSLIRIO DA SILVA JUNIOR

Localização de Defeitos Evolucionária Baseada em Fluxo de Dados

Goiânia
2020

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO (TECA) PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES

E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo das Teses e Dissertações disponibilizado na BDTD/UFG é de responsabilidade exclusiva do autor. Ao encaminhar o produto final, o autor(a) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do material bibliográfico

[x] Dissertação [ ] Tese

2. Nome completo do autor

Deuslírio da Silva Júnior

3. Título do trabalho

Localização de Defeitos Evolucionária Baseada em Fluxo de Dados

4. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador)

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante:
a) consulta ao(à) autor(a) e ao(à) orientador(a);
b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo da tese ou dissertação.
O documento não será disponibilizado durante o período de embargo.
Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

Obs. Este termo deverá ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Plinio De Sa Leitão Junior**, **Professor do Magistério Superior**, em 24/08/2020, às 11:16, conforme

horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **DEUSLIRIO DA SILVA JUNIOR**, **Discente**, em 24/08/2020, às 11:37, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1505201** e o código CRC **8F7401E8**.

Referência: Processo nº 23070.028166/2020-14 SEI nº 1505201

DEUSLIRIO DA SILVA JUNIOR

Localização de Defeitos Evolucionária Baseada em Fluxo de Dados

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Plínio de Sá Leitão Júnior

Goiânia
2020

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Silva-Junior, Deuslirio
Localização de Defeitos Evolucionária Baseada em Fluxo de Dados [manuscrito] / Deuslirio Silva-Junior. - 2020.
LXXIX, 79 f.: il.

Orientador: Prof. Dr. Plinio S. Leitao-Junior.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2020.
Bibliografia. Apêndice.
Inclui gráfico, tabelas, lista de figuras, lista de tabelas.

1. Localização de defeitos. 2. SBSE. 3. Depuração de Software. 4. Fluxo de dados. I. S. Leitao-Junior, Plinio, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE DISSERTAÇÃO

Ata nº **13/2020** da sessão de Defesa de Dissertação de **Deuslírio da Silva Júnior**, que confere o título de Mestre Ciência da Computação, na área de concentração em Ciência da Computação.

Aos vinte e dois dias do mês de julho de dois mil e vinte, a partir das nove horas e quarenta minutos, via sistema de webconferência da RNP, realizou-se a sessão pública de Defesa de Dissertação intitulada "**Localização de Defeitos Evolucionária Baseada em Fluxo de Dados**". Os trabalhos foram instalados pelo Orientador, Professor Doutor Plínio de Sá Leitão Júnior (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Marcos Lordello Chaim (EACH/USP), membro titular externo; e Professora Doutora Telma Woerle de Lima Soares (INF/UFG), membra titular interna. A realização da banca ocorreu per meio de videoconferência, em atendimento à recomendação de suspensão das atividades presenciais na UFG emitida pelo Comitê UFG para o Gerenciamento da Crise COVID-19, bem como à recomendação de isolamento social da Organização Mundial de Saúde e do Ministério da Saúde para enfrentamento da emergência de saúde pública decorrente do novo coronavírus. Durante a arguição os membros da banca não fizeram sugestão de alteração do título do trabalho. A Banca Examinadora reuniu-se em sessão secreta a fim de concluir o julgamento da Dissertação, tendo sido o candidato **aprovado** pelos seus membros. Proclamados os resultados pelo Professor Doutor Plínio de Sá Leitão Júnior, Presidente da Banca Examinadora, foram encerrados os trabalhos e, para constar, lavrou-se a presente ata que é assinada pelos Membros da Banca Examinadora, aos vinte e dois dias do mês de julho de dois mil e vinte.

TÍTULO SUGERIDO PELA BANCA

Documento assinado eletronicamente por **Telma Woerle De Lima Soares**, **Professora do Magistério Superior**, em 22/07/2020, às 12:08, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Plinio De Sa Leitão Junior**, **Professor do Magistério Superior**, em 22/07/2020, às 12:08, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcos Lordello Chaim**, **Usuário Externo**, em 22/07/2020, às 12:09, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **DEUSLIRIO DA SILVA JUNIOR**, **Discente**, em 22/07/2020, às 12:26, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?

acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1403909** e o código CRC **36873586**.

Referência: Processo nº 23070.028166/2020-14 SEI nº 1403909

Deuslirio da Silva Junior

Graduou-se em Ciência da Computação pela UFG - Universidade Federal de Goiás. Durante a graduação foi monitor voluntário, participou do projeto de extensão GIAD - Grupo Integrado de Ações Contra a Dengue da UFG, e desenvolveu atividades como colaborador no Media Lab/UFG - Laboratório de Pesquisa, Desenvolvimento e Inovação em Mídias Interativas. Durante o mestrado foi bolsista CAPES e integrando o grupo de pesquisa I4Soft - Intelligence for Software conduziu investigações à respeito de Localização de Defeitos Baseada em Busca

Dedico este trabalho principalmente à minha família, e aos meus amigos.

Agradecimentos

Agradeço aos meus amigos e guias pelo apoio e iluminação nas horas difíceis, em especial o Eduardo Faria de Souza e o Mateus Machado Luna pelas tardes de reflexão.

À minha família, que sempre contribuiu para a minha formação e apoiou as minhas decisões. Agradeço principalmente à minha mãe por todo o esforço e dedicação que sempre empregou na minha criação, e ao meu pai, pelo apoio e suporte que foram essenciais para mim e para a conclusão deste trabalho.

Pelo carinho, compreensão e companheirismo durante todos esses anos, agradeço à Jessika Nunes Caetano, que me acompanha e me fortalece no nosso caminho. Agradeço também a família dela, que me acolheu e também torce pelo meu sucesso.

Ao meu orientador Plínio de Sá Leitão Júnior, pelos ensinamentos e direcionamentos na condução deste trabalho. Agradeço também todos os membros do laboratório 254 por compartilharem comigo esse período de tanto aprendizado e crescimento, em especial ao Diogo de Freitas, Lucas Pacífico, Sávio Sampáio, Marcos Vinícius Silva, e o professor Altino Dantas que esteve sempre ao meu lado, pronto para compartilhar os seus conhecimentos.

Aos técnicos administrativos, professores e toda a equipe de limpeza e manutenção que trabalham para manter o instituto de informática um espaço onde ideias e iniciativas podem ser desenvolvidas com bastante qualidade.

Aos professores Valdemar Graciano Neto, Celso Camilo Júnior, Rachel Harrison, e Marcos Lordello Chaim pelo suporte, atenção e ensinamentos nas conduções e escrita dos trabalhos.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio e financiamento da pesquisa. E aos professores que compõem a banca examinadora, pela atenção e contribuições à este trabalho.

The greatest shortcoming of the human race is our inability to understand the exponential function.

Albert Allen Bartlett,
Arithmetic, Population and Energy.

Resumo

Silva-Junior, D.. **Localização de Defeitos Evolucionária Baseada em Fluxo de Dados**. Goiânia, 2020. 90p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Contexto - Localização de defeitos (*Fault Localization*) é descrita como o processo de precisamente indicar o(s) elemento(s) que torna(m) defeituoso um dado código de programa. Esta é uma atividade conhecida por ser demasiadamente custosa e monótona. Automatizar esse processo têm sido o objetivo de diversos estudos, tendo se mostrado um problema desafiador. Uma estratégia comum é associar um valor de propensão a defeito a cada comando do código. A maioria dos métodos que utilizam esta estratégia, são heurísticas que usam como fonte de informação os comandos executados durante o teste de software. Estas abordagens são conhecidas por serem baseadas no espectro de cobertura de fluxo de controle (control-flow). **Objetivo** - O presente estudo busca investigar outra fonte de informação sobre os defeitos, o fluxo de dados (data-flow), que se expressa pela relação de locais de definição e locais uso de variáveis. Como o fluxo de dados pode contribuir para a localização de defeitos e como utilizá-lo em estratégias evolucionários são interesses deste trabalho. **Abordagem** - São apresentadas duas abordagens evolucionárias, uma baseada em algoritmo genético (GA) que busca combinar diferentes heurísticas utilizando fluxo de controle e fluxo de dados como fonte de informação sobre defeitos. E outra baseada em programação genética (GP), que utiliza de novas variáveis que expressam o espectro de cobertura de fluxo de dados, para gerar novas equações, vocacionadas à localização de defeitos. **Resultados** - A abordagem GA foi avaliada em 7 programas C pequenos que compõem o *Siemens Suite*, *benchmark* bastante utilizado em abordagens similares, e em um conjunto de versões defeituosas do programa *jsoup*. As métricas de avaliação utilizadas descrevem a eficácia do ponto de vista absoluto, e também a dependência por estratégias de desempate. Nesse contexto, apesar de a abordagem utilizando apenas fluxo de dados produzir resultados competitivos, a abordagem híbrida (fluxo de controle e fluxo de dados) se destaca por manter bons resultados quanto a eficácia, e ainda ser menos dependente de desempates. A abordagem GP por sua vez foi investigada quanto à eficácia utilizando métricas populares neste contexto, e também quanto a eficiência, através da contagem dos ciclos de execuções (gerações) necessários para apresentar resultados competitivos. Novamente a estratégia híbrida se destaca por produzir os mesmos resultados que outros métodos, mas demandando menos gerações para isso. **Conclusões** - Os resultados de ambas as abordagens destacam que apesar de fluxo de dados apresentar boa eficácia para a localização de defeitos, as estratégias híbridas, utilizando fluxo de controle e fluxo de dados como fontes de informação sobre defeitos, superam todas os métodos usados como comparação. Porém mais investigações devem ser conduzidas em diferentes conjuntos de programas.

Palavras–chave

Localização de defeitos, Depuração de software, SBSE, Fluxo de dados

Abstract

Silva-Junior, D.. **Data-flow-based evolutionary fault localization**. Goiânia, 2020. 90p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Context - Fault localization is the activity of precisely indicating the faulty commands in a buggy program. This is an activity known to be too costly and monotonous. Automating this process has been the objective of several studies, having proved to be a challenging problem. A common strategy is to associate a suspiciousness value to each command in the code. Most methods, which use this strategy, are heuristics that use the commands executed during the software test as an information source. These approaches are known to be based on the control-flow coverage spectrum. **Objective** - The present study seeks to investigate another source of information about faults, the data-flow, which is expressed by the relationship between the places of definition and places of use of variables. How the data-flow can contribute to fault localization and how to use it in evolutionary strategies are interests of this work. **Approach** - Two evolutionary approaches are presented, one based on a genetic algorithm (GA) that seeks to combine different heuristics using control-flow and also data-flow as a sources of information about faults. And another, based on genetic programming (GP), which uses new variables that express the data-flow coverage spectrum, to generate new equations, more fitted to fault localization. **Results** - The GA approach was evaluated in 7 small C programs that make up the *Siemens Suite*, *benchmark* widely used in similar approaches, and also in a set of faulty versions of the Java program *jsoup*. The evaluation metrics used describe the effectiveness from an absolute point of view, as well as the dependence on tiebreak strategies. In this context, although the approach using only data-flow produces competitive results, the hybrid approach (control-flow and data-flow) stands out for maintaining good results in terms of effectiveness, and still being less dependent on tiebreakers. The GP approach in turn was investigated for effectiveness using popular metrics in this context, and also for efficiency, by counting the cycles of executions (generations) necessary to present competitive results. Again, the hybrid strategy stands out for producing the same results as other methods, but requiring less generations to do so. **Conclusions** - The results of both approaches highlight that although data-flow has good effectiveness in locating defects, hybrid strategies, using control- and data-flow as sources of information about defects, outperforms all the methods used as a comparison. However, further investigations must be conducted in different sets of programs.

Keywords

Fault localization, Software debugging, SBSE, Data-flow analysis

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

Software vem sendo cada vez mais adotado para gerir, executar e automatizar tarefas dos mais diversos contextos, inclusive tarefas críticas como em sistemas relacionados à saúde e segurança. Por outro lado, defeitos são tão comuns no ciclo de desenvolvimento de software, que boa parte dos custos de produção estão diretamente relacionados à prevenção e correção dos mesmos. Devido ao alto custo de prevenção e correção de defeitos e ao alto potencial de danos, a presença de defeitos representa riscos importantes quanto a recursos e a vidas humanas [13].

Teste e depuração de software, dentre outros objetivos, visam aumentar e garantir a confiabilidade de artefatos de software. Fazem parte desse contexto as tarefas de prevenir, localizar e reparar defeitos de software. [5]. Quando a atividade de teste revela a presença de defeitos em software, em geral três etapas são pertinentes à depuração e consequente solução do problema: os locais dos defeitos devem ser encontrados no código; os defeitos devem ser compreendidos; e finalmente os defeitos devem ser corrigidos [27].

Uma importante etapa no processo de depuração de software é a chamada Localização de Defeitos (*Fault Localization – FL*), que pode ser definida como *o processo de precisamente indicar o(s) elemento(s) que torna(m) defeituoso um dado código de programa*. Esta é uma atividade conhecida por engenheiros de software por ser custosa e monótona. Justamente por impactar diretamente no ciclo de desenvolvimento de software, automatizar esse processo têm sido o objetivo de diversos estudos, e tem se mostrado um problema desafiador [38].

Uma estratégia, comum na literatura, na tentativa de localizar defeitos automaticamente, é associar um valor (número) a cada comando do código: o valor de propensão a defeito (ou valor de suspeita) que o comando possui. É esperado que comandos com valores altos de propensão a defeito, tenham maior chance de serem defeituosos.

A maioria dos métodos de localização de defeitos, que procuram atribuir valores de suspeita aos comandos, são heurísticas que usam informação de comandos executados ao se aplicar determinado conjunto de casos de teste, para determinar o quão propenso a defeitos cada comando é [38]. Estas abordagens são conhecidas por serem baseadas no espectro de cobertura de fluxo de controle [4].

O espectro de cobertura de fluxo de controle tem sido amplamente explorado em trabalhos relacionados à localização de defeitos em software [2, 3, 18, 19, 26, 36]. O fluxo de controle oferece uma granularidade muito próxima da esperada pela estratégia de atribuição de suspeita, já cada comando, ou bloco de comando, pode ser visto como um nó do grafo de fluxo de controle [11]. Porém, é possível obter espectro de cobertura de outras fontes, como por exemplo o *espectro de fluxo de dados*.

Diferentemente do fluxo de controle, a cobertura de fluxo de dados pela execução de casos de teste é descrita sob a ótica de associações entre a definição e o uso de variáveis (associação def-uso): comandos de atribuição de valor e comandos de uso desse valor, respectivamente. A literatura apresenta inciativas de pesquisa em que o cálculo de propensão a defeito emprega variáveis oriundas de cobertura associações def-uso [24, 30, 31].

Recentemente, estratégias baseadas em algoritmos de busca, tais como *Programação Genética* (*Genetic Programming*), *Têmpera Simulada* (*Simulated Annealing*) e *Algoritmo Genético* (*Genetic Algorithm*), têm ganhado destaque em localização de defeitos [22]. Sendo principalmente utilizadas para combinar e/ou gerar heurísticas que, quando aplicadas às informações de cobertura de fluxo de controle, produzem resultados que não seriam trivialmente alcançados por métodos determinísticos.

O problema investigado na presente pesquisa é mostrado a seguir.

Contexto. Métodos para a automação da localização de defeitos são continuamente propostos, pois esta é uma atividade custosa e ininterrupta durante a 'vida do software'. Nessa perspectiva, o termo 'métodos tradicionais' se refere aos métodos alicerçados em espectro de cobertura conhecidos até então, a saber: os *determinísticos* baseados em informação de fluxo de controle e/ou fluxo de dados; e os *evolucionários* baseados em fluxo de controle.

Problema: Métodos evolucionários para a localização de defeitos, que empregam cobertura de fluxo de dados como fonte de informação sobre defeitos, obtêm resultados competitivos em relação aos 'métodos tradicionais'?

Justificativa: A razão para tal questão é a investigação se a conjugação de dois objetos – meta-heurísticas evolucionárias e informação de fluxo de dados – que isoladamente fundamentam métodos exitosos à localização de defeitos, é promissora para a proposição de novas abordagens; *i.e.* se abordagens que se baseiam em ambos objetos são competitivas.

1.1 Objetivos

O *objetivo principal* desta pesquisa é lidar com o problema posto acima, ou seja, buscar resolvê-lo pela proposição de abordagens em sintonia com a conjunção dos objetos do problema, bem como analisar tais propostas quanto à efetividade para a localização de defeitos. Usamos a premissa de que o espectro de fluxo de dados

não é devidamente explorado pelos métodos de localização de defeitos, conforme será apresentado no Capítulo 2.

Abstraímos alguns *objetivos específicos* que julgamos pertinentes para alcançar o objetivo principal. Cada *objetivo específico* busca solucionar um subproblema, conforme abaixo:

- que meta-heurísticas evolucionárias são aplicadas ao problema, conforme pesquisas existentes na área?
- que variáveis traduzem a essência da cobertura de fluxo de dados?
- como construir equações para o cálculo de suspeita baseada em variáveis de fluxo de dados?
- como atribuir valores de suspeita às associações de fluxo de dados?
- como calcular valores de suspeita dos comandos (nós) do programa a partir dos valores de suspeita das associações de fluxo de dados?
- como lidar com valores de suspeita 'empatados' com respeito a vários comandos (nós) do programa?
- que *benchmarks* e *baselines* são pertinentes à avaliação das abordagens propostas?
- como medir e avaliar as abordagens propostas?

1.2 Principais Contribuições

Durante a condução das atividades no escopo deste trabalho foram produzidas diversas proposições e artefatos, algumas destas já foram publicadas no *The 35th ACM/SIGAPP Symposium On Applied Computing* em artigo intitulado *Data-Flow-Based Evolutionary Fault Localization* [33]. Outras ainda serão melhor exploradas em futuras investigações. Assim, são destaques como contribuições deste trabalho:

- Estratégia para o cálculo de propensão a defeitos para comandos (nós) a partir da análise de fluxo de dados;
- Abordagem evolutiva utilizando *Algoritmo Genético* para a combinação de diferentes heurísticas, pelo emprego dos espectros de fluxo de controle e de dados, isoladamente ou em conjunto;
- Definição de oito novas variáveis que expressem o espectro de fluxo de dados, das perspectivas das análises de definição e de uso de dados;
- Abordagem evolutiva utilizando *Programação Genética*, para gerar novas heurísticas, cujas equações incluem as oito novas variáveis de cobertura de fluxo de dados, e as quatro variáveis tradicionais de cobertura de fluxo de controle;
- Métrica de avaliação que expressa a dependência de estratégias de desempate para métodos de localização de defeitos, de um ponto de vista absoluto;

- Avaliação de abordagens de localização de defeitos baseadas em busca, pelas análises da eficácia (habilidade para localizar defeitos e da eficiência número de ciclos de execução (gerações).

1.3 Organização do trabalho

A divisão deste trabalho é definida da seguinte forma. O Capítulo 2 dedica-se a discutir os fundamentos necessários e trabalhos relacionados úteis para o entendimento do trabalho como um todo, abordando teste e depuração de software, fluxo de controle e fluxo de dados, localização de defeitos baseada em busca, computação evolucionária, e ainda uma contextualização das métricas de avaliação utilizadas por pesquisas no mesmo tópico.

No Capítulo 3 são apresentadas duas abordagens evolucionárias a primeira utilizando da meta-heurística algoritmo genético, e combinando os valores obtidos por diversas heurísticas tradicionais, aplicadas tanto em análise de fluxo de controle, quanto em análise de fluxo de dados. A segunda abordagem valendo do potencial da programação genética, objetiva a geração de novas heurísticas para a localização de defeitos, para isso, dispõe de oito novas variáveis que são utilizadas isoladamente e também em conjunto com as quatro variáveis de cobertura tradicionais.

No Capítulo 4 são discutidos os processos de experimentação, *benchmarks*, *baselines*, parâmetros dos algoritmos e ferramentas utilizadas. Apresentando ainda resultados obtidos por ambas as abordagens, e aspectos considerados limitações ou ameaças à validade dos experimentos.

Por último, o Capítulo 5 discute alguns pontos encontrados durante esta pesquisa que merecem destaque, além de trazer as conclusões e perspectivas de possíveis desdobramentos deste trabalho.

CAPÍTULO 2

Fundamentação Teórica

Este capítulo apresenta os conceitos pertinentes ao entendimento das demais seções deste trabalho. A Seção 2.1 aborda *Teste de Software* e como os elementos de programa analisados no teste podem contribuir para a *Localização de Defeitos* de software. A Seção 2.2 explora *Espectros de Dados* sobre defeitos e *Métricas de Avaliação* utilizadas na automatização do processo de localização de defeitos em software. A Seção 2.3 é dedicada a apresentar computação evolucionária e como funciona algumas abordagens para problemas de busca. Finalmente a Seção 2.4 discute como o problema de localização de defeitos pode ser encarado pela ótica da otimização baseada em busca.

2.1 Teste de Software

Teste de Software consiste na verificação dinâmica de que um programa fornece comportamentos esperados em um conjunto finito de casos de teste, adequadamente selecionados no domínio de execução geralmente infinito [34]. Para isso o software é executado com entradas predefinidas, verificando se a saída obtida está de acordo com o a saída esperada. Se essa confirmação não acontece, pode-se afirmar que uma falha foi identificada. Nesse sentido, o teste é uma ferramenta para alcançar maior confiança no funcionamento do software.

Além de servirem para indicar a presença de um defeito, as informações de execução do teste também podem servir como apoio para a localização e a correção do defeito encontrado [11]. Os autores destacam que a atividade de teste pode ser dividida em fases com diferentes objetivos, sendo elas:

- Teste de Unidade: Aborda as menores unidades do programa, como funções, procedimentos, classes ou métodos. Assim, erros relacionados a algoritmos e estruturas de dados podem ser melhor identificados. Esse tipo de teste pode ser aplicado conforme as funcionalidades são implementadas, sem a necessidade do sistema completo estar finalizado.

- Teste de Integração: Deve ser realizado após o teste individual das unidades. Foca em verificar o funcionamento das partes do software trabalhando juntas, para garantir que a interação entre as unidades não leve a algum comportamento inadequado.
- Teste de Sistemas: Utilizado após o sistema estar completo e totalmente integrado. O objetivo é garantir que o software está de acordo com todas as funcionalidades descritas nos documentos de requisitos, bem como os requisitos não funcionais.
- Teste de Regressão: Ao modificar um sistema corre-se o risco de introduzir novos defeitos, por esse motivo este tipo de teste é realizado na etapa de manutenção do software. Servindo para garantir que a implementação de novos requisitos não façam funcionalidades já validadas passarem a ter um comportamento diferente do esperado.

2.1.1 Cobertura de elementos de programa

Critérios de Teste definem requisitos à atividade de teste; ou seja, estabelecem elementos requeridos que orientam o teste. Nesse sentido, *Grafos Direcionados* representam abstrações à aplicação de critérios de teste. Segundo Ammann e Offutt [5], grafos direcionados são úteis para critérios em dois tipos: os primeiros são geralmente referidos como *critérios de cobertura de fluxo de controle*; os últimos são baseados no fluxo de dados através do artefato de software representado pelo grafo e são chamados de *critérios de cobertura do fluxo de dados*.

Como exemplo, considere o Código de Programa 2.1; daqui em diante usaremos esse código para melhor esclarecer os conceitos apresentados. Critérios baseados em grafos exigem que o teste cubra o grafo de diferentes maneiras [5].

Código 2.1 Código exemplo

```
1  void mid(int x, int y, int z ) {
2      int m = z;          //0
3      if (y<z){           //1
4          if (x<y){       //2
5              m = y;      //5
6          }else{
7              if (x<z){   //4
8                  m = x;  //9
9              }
10         }
11     }else{
12         if (x>y){       //3
13             m = y;      //7
14         }else{
15             if (x>z){   //6
16                 m = x;  //11
17             }
18         }
19     }
20     printf("%d",m);     //12
21 }
```

Fluxo de Controle

Grafo de Fluxo de Controle (GFC) é a representação de um programa *P*, onde: cada *nó* (*node*) equivale a um bloco de comandos; cada *aresta* (*edge*) simboliza um possível desvio de um bloco para outro. Convencionalmente, um GFC possui um único *nó de entrada* (vértice de entrada), diversos nós internos e apenas um *nó de saída* (vértice de saída) [11]. Dessa forma, qualquer execução do programa pode ser interpretada como a cobertura de uma sequência finita de nós, ou seja, um *caminho completo* do nó de entrada ao nó de saída percorrido no GFC. A Figura 2.1 apresenta a representação GFC do código exemplo apresentado anteriormente.

Critérios de Cobertura de Grafos determinam que elementos do grafo devem ser 'exercitados'. Especificamente, *Critérios Baseados em Fluxo de Controle* requerem que, durante o teste, haja a cobertura de elementos do GFC, tais como nós, arcos e caminhos.

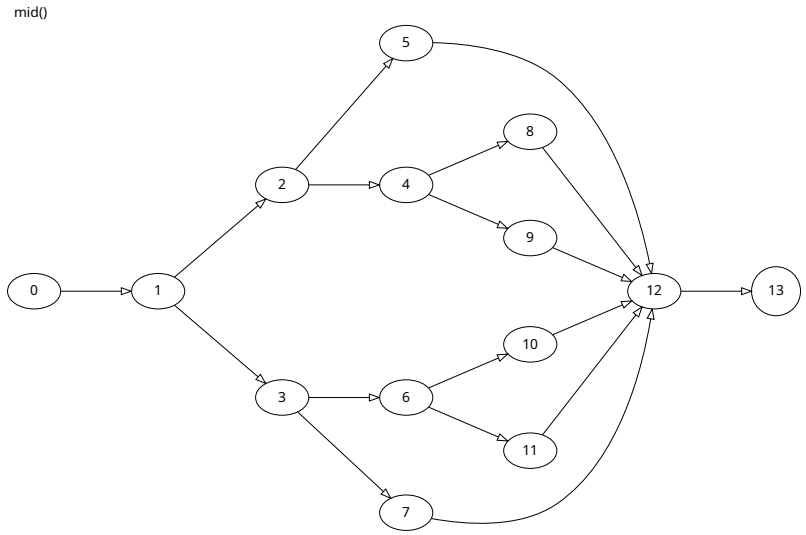

Figura 2.1: Grafo de fluxo de controle.

Fluxo de Dados

Critérios de Fluxo de Dados são baseados na premissa que, para testar um software, deve-se garantir que cada variável criada em um determinado local do código vai ser definida e usada apropriadamente. Isso pode ser feito focando na definição e uso dos dados [5]. Em um código, uma *definição* (*def*) é o local em que uma variável foi escrita em memória, e um *uso* (*use*) acontece quando esse 'local em memória' é acessado. Quanto ao uso é possível distinguir dois tipos, *c-uso* (uso computacional) que afeta diretamente uma computação da variável em questão, e o *p-uso* (uso predicativo) que afeta o fluxo de controle do programa. Dessa forma, é convencionado que *c-usos* acontecem em nós do GFC e *p-usos* em arestas [11], como podemos ver na Figura 2.2.

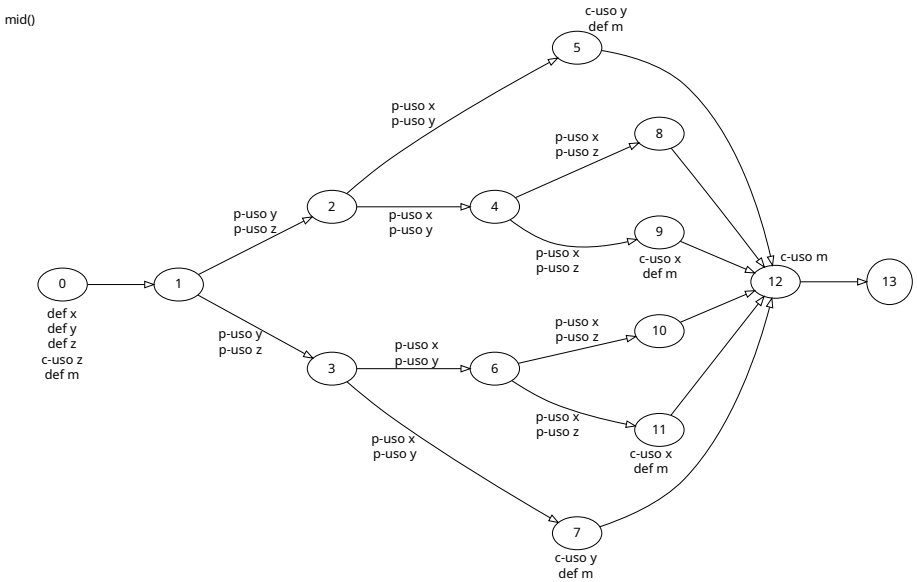

Figura 2.2: Grafo de fluxo de dados.

Quando há uma definição de variável e subsequente uso da mesma – um caminho no GFC entre o local de definição e o local de uso, sem haver a re-definição da variável nesse caminho – é caracterizada uma *Associação de Fluxo de Dados*, ou associação def-uso (*dua – Definition-Use Association*). Critérios de fluxo de dados envolvem o exercício de *dua*'s de diversas maneiras [5], tais como executar qualquer *caminho livre de definição* entre os locais de definição e de uso, ou executar um caminho específico entre esses locais.

Uma *dua* pode ser descrita como uma tripla da forma: <Variável, nó de def, nó ou arco de uso> a Tabela 2.1 apresenta todas as *dua*'s que podem ser observadas no grafo apresentado da Figura 2.2.

Tabela 2.1: Todas as *dua*'s extraídas do grafo na Figura 2.2.

Variável	Definição	Uso	Variável	Definição	Uso
z	0	0	y	0	5
y	0	1 - 2	y	0	1 - 3
z	0	1 - 2	z	0	1 - 3
y	0	2 - 5	y	0	1 - 2
x	0	2 - 5	x	0	2 - 4
z	0	4 - 9	z	0	4 - 8
x	0	4 - 9	x	0	4 - 8
x	0	9	y	0	7
y	0	3 - 6	y	0	3 - 7
x	0	3 - 6	x	0	3 - 7
x	0	6 - 10	x	0	6 - 11
z	0	6 - 10	z	0	6 - 11
x	0	11	m	0	12
m	5	12	m	7	12
m	9	12	m	11	12

2.2 Localização de Defeitos em Software

Localização de Defeitos (FL – Fault Localization) refere-se à localização de elementos de software defeituosos relacionados a falhas produzidas durante a execução de caso de teste: esta é uma tarefa trabalhosa e demorada [22].

A automatização do processo de FL tem sido objeto de estudo de diversos trabalhos nos últimos anos [38]. Com o objetivo de indicar com precisão o local do *bug* em programas defeituosos, algumas vertentes se destacam, tal como a localização de defeitos baseada em *espectro de cobertura*; dois exemplos são o *espectro de fluxo de controle e o espectro de fluxo de dados. Nesse sentido, a literatura pontua que,* ao se aplicar técnicas de busca alinhadas ao espectro de fluxo de controle, é possível alcançar resultados que dificilmente seriam obtidos por métodos tradicionais [40].

2.2.1 Espectro de fluxo de controle

As principais propostas de automatização de FL se baseiam na execução dos casos de teste, para então calcular uma medida que traduz a probabilidade de cada

elemento de programa ser defeituoso. O *espectro de cobertura* é definido como um conjunto de dados que expressam o comportamento de execução do programa. Nesse sentido ele permite a visualização de quais elementos de programa são executados (cobertos) pelos casos de teste e como eles podem estar relacionados ao defeito.

O espectro de fluxo de controle é definido como uma matriz $M \times N$, obtida a partir de M execuções que podem cobrir N elementos de programa: a informação de sucesso ou falha da execução pode ser representada como uma coluna à parte da matriz. A partir dos dados desta matriz de cobertura, é possível medir a propensão a defeito (ou suspeita) S para qualquer elemento N_i [4], como pode ser visto na Figura 2.3.

$$\begin{array}{cc} & \overbrace{\qquad\qquad}^{N \text{ elementos}} \\ M \text{ execuções} & \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \end{array} \quad \begin{array}{c} \text{falhas} \\ \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix} \end{array}$$

$$S(N_1) \quad S(N_2) \quad \cdots \quad S(N_n)$$

Figura 2.3: Exemplo de matriz de fluxo de controle.

As heurísticas de localização de defeitos baseadas em espectro de fluxo de controle (SBFL – *Spectrum-based fault localization*), são equações que utilizam os valores de variáveis obtidas da matriz de fluxo de controle para atribuir aos elementos de programa um valor de propensão a ser o elemento defeituoso. Utilizaremos a seguinte notação para indicar estas variáveis, que podem ser obtidas para cada elemento de programa N_i:

- es: Número de casos de teste positivos que executam o elemento observado.
- ns: Número de casos de teste positivos que não executam o elemento observado.
- ef: Número de casos de teste negativos que executam o elemento observado.
- nf: Número de casos de teste negativos que não executam o elemento observado.

A heurística *Tarantula* [19] foi uma das primeiras técnicas de localização de defeitos baseada em espectro de fluxo de controle apresentadas. Como visto anteriormente, um elemento de programa, tal como comando (ou bloco), pode ser representado por um nó do grafo de fluxo de controle. A heurística *Tarantula* "pode medir" a propensão a defeito de um dado nó do programa investigado, como apresentado na Equação 2-1.

$$Tarantula(node) = \frac{\dfrac{ef(node)}{ef(node)+nf(node)}}{\dfrac{ef(node)}{ef(node)+nf(node)} + \dfrac{es(node)}{es(node)+ns(node)}} \tag{2-1}$$

Tabela 2.2: Exemplo de Localização de defeitos baseada em espectro de cobertura.

Linha	Código	tc1 (1,1,3)	tc2 (1,2,3)	tc3 (3,2,1)	tc4 (3,3,3)	tc5 (3,1,2)	tc6 (2,1,3)	Tarantula
0	void mid(int x, int y, int z){	•	•	•	•	•	•	0.5
1	**int m = x;** /*m=z*/	•	•	•	•	•	•	0.5
2	if (y<z){	•	•	•	•	•	•	0.5
3	if (x<y){	•	•			•	•	0.62
4	m = y;		•					0
5	}else{							0
6	if (x<z){	•				•	•	**0.71**
7	m = x;	•					•	0
8	}							0
9	}							0
10	}else{							0
11	if (x>y){			•	•			0
12	m = y;			•				0
13	}else{							0
14	if (x>z){				•			0
15	m = x;							0
16	}							0
17	}							0
18	}							0
19	printf("%d",m);	•	•	•	•	•	•	0.5
20	}							0
	Sucesso / Falha	S	S	S	S	F	S	

Os resultados obtidos por esta heurística podem ser utilizados como um “guia” para a localização de defeitos. Assim, diversas outras heurísticas passaram a ser apresentadas objetivando uma melhor precisão. Além da *Tarantula*, diversas outras heurísticas são utilizadas como métodos para a localização de defeitos, a Tabela 2.3 apresenta algumas destas heurísticas. Trabalhos como *Ochiai* [1], *Dstar* [36], e *OP2* [26] e se mostraram eficientes quanto à proporção do número de comandos investigados até encontrar o local do defeito, em relação ao número total de elementos.

Mais recentemente a preocupação quanto à localização de defeitos se voltou para métricas absolutas [27], que avaliam a eficiência dos métodos quanto ao se investigar um número fixo de elementos. Como consequência, métodos mais complexos se destacaram [7, 35, 40]. Ainda assim, os resultados obtidos por estes métodos não são eficazes para todos os casos.

As técnicas de localização de defeitos em software enfrentam diversas dificuldades, além de destacar o comando defeituoso, elas também devem evitar que comandos que não tenham relação com o defeito, sejam indicados com alto valor de suspeita, isto é, técnicas de localização de defeitos não devem atribuir alta suspeita a elementos que não sejam defeituosos.

Tabela 2.3: Exemplos de heurísticas para a localização de defeitos.

Heurística	Nome	Equação	Heurística	Nome	Equação
H_1	Tarantula	$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{es}{es+ns}}$	H_7	GP13	$ef \times \left(1+\frac{1}{2\times es+ef}\right)$
H_2	Ochiai	$\frac{ef}{\sqrt{ef+nf\times(ef+es)}}$	H_8	OP2	$\frac{ef-es}{es+ns+1}$
H_3	OP	$\begin{cases} nf>0 & -1 \\ nf\leq 0 & ns \end{cases}$	H_9	Wong3	$\begin{cases} es\leq 2 & ef-es \\ es>2 \ \& \ es\leq 10 & ef-2+0.1\times(es-2) \\ es<10 & ef-2.8+0.01\times(es-10) \end{cases}$
H_4	Dstar	$\frac{ef^2}{nf+es}$	H_{10}	Zoltar	$\frac{ef}{ef+nf+es+\frac{10000\times nf\times es}{ef}}$
H_5	Ample	$\left\lvert\frac{ef}{ef+nf}-\frac{es}{es+ns}\right\rvert$	H_{11}	Kulczynski2	$\frac{1}{2}\times\left(\frac{ef}{ef+nf}+\frac{ef}{ef+es}\right)$
H_6	Jaccard	$\frac{ef}{ef+nf+es}$	H_{12}	Barinel	$1-\frac{es}{es+ef}$

2.2.2 Espectro de fluxo de dados

Conforme ressaltado na Equação 2-1, que define a heurística *Tarantula*, o espectro de fluxo de controle é materializado pelas variáveis $ef(node)$, $es(node)$, $nf(node)$ e $ns(node)$, as quais sumarizam o comportamento de fluxo de dados, quanto à cobertura e ao resultado do teste, para um nó específico (no caso o nó denominado *node*).

O *espectro de fluxo de dados* é caracterizado de maneira similar, contudo refere-se à cobertura de associações de fluxo de dados (cobertura de *duas*), em vez de cobertura de nós. Dessa forma, o espectro é concretizado pelas variáveis $ef(dua)$, $es(dua)$, $nf(dua)$ e $ns(dua)$, que resumem o comportamento de fluxo de dados para uma associação específica (no caso a associação é denominada *dua*). A Subseção 3.2.1 possui maiores informações sobre as varáveis de fluxo de dados e suas variações. No restante da presente subseção, são mostrados dois trabalhos que exploram o espectro de fluxo de dados para a localização de defeitos.

Santelices *et al.* [31] avaliam localização de defeitos utilizando diferentes tipos de elementos de programas. Explorando o espectro de cobertura aplicado a comandos (*statements*), estruturas de decisão (*branches*), e associações def-uso (*dua*'s). O estudo mostra que nenhum tipo de cobertura exprime a melhor performance em todos os defeitos analisados, diferentes tipos de defeitos são melhor localizados por diferentes tipos de cobertura. A análise das *dua*'s não distingue *p-usos* de *c-usos*, ou seja, mesmo que o uso da variável seja em uma aresta do GFC, ele é interpretado como ocorrendo apenas no primeiro nó. O estudo ainda destaca o custo computacional de se obter as *dua*'s exercitadas por um programa, e traz uma comparação de resultados utilizando a cobertura de associações de fluxo de dados inferidas a partir da cobertura de *branches*.

Para tornar possível a comparação entre cobertura de *branches*, *dua*'s e *state-*

ments, os autores de [31] apresentam uma forma de mapear a propensão a defeito obtida por *branches* e *dua*'s aos comandos que as compõem. Isto acontece através de três regras.

1. Cada *branch* é associada ao comando de sua condicional e cada *dua* é associada ao seu comando de definição.
2. Cada *branch* e *dua* é associada a todos os comandos que precedem a sua condicional, ou definição, no mesmo bloco de comandos.
3. Após aplicar as regras 1 e 2 se algum comando não foi mapeado a *branches* ou *dua*'s, ele é associado a todas as *branches* que ele é dependente, e às *dua*'s onde este comando aparece com uso.

A abordagem é avaliada em 107 versões defeituosas. Versões estas obtidas de seis programas do conjunto *Siemens Suite* [17] que foram traduzidas de C para Java (o maior programa do conjunto, o *replace*, não foi utilizado), e outros oito programas que não precisaram de tradução. A propensão a defeito utilizada na análise foi obtida apenas para a heurística *Tarantula*.

Como nesse conjunto de programas nenhum tipo cobertura se destacou em todos os defeitos, os autores propõem a combinação das diferentes propensões, definindo as funções *max-SBD* (*Max-Statement-Branch-DU-Pair*) que utiliza a máxima propensão obtida por comandos, *branches* ou *dua*'s, *avg-SBD* (*Average-Statement-Branch-DU-Pair*) que utiliza a média das propensões obtidas pelos mesmos três elementos da anterior e *avg-BD* (*Average-Branch-DU-Pair*) que utiliza a média de das propensões obtidas apenas por *branches* e *dua*'s.

De acordo com os resultados obtidos, as métricas *avg-SBD* e *avg-BD* apresentam os melhores resultados no conjunto de defeitos utilizado. A performance das métricas é avaliada pelo chamado custo de localização de defeito, que se refere à posição do comando defeituoso dividida pelo número total de elementos investigados para aquela versão defeituosa. O trabalho ainda destaca que inferindo a cobertura de *dua*'s pela cobertura de *branches*, o resultado é tão representativo quanto utilizando as *dua*'s diretamente, porém com menor custo computacional para ser obtida.

A ferramenta de código aberto **Jaguar** (*JAva coveraGe faUlt locAlization Ranking*) [29] pode ser utilizada para indicar não apenas as linhas mais prováveis a defeitos, mas também as *dua*'s mais prováveis, ou seja, a ferramenta suporta análise de cobertura utilizando tanto fluxo de controle quanto utilizando fluxo de dados. A ferramenta Jaguar é composta por duas partes a *Jaguar Runner* que é encarregada de coletar e gerar a lista de elementos de programa, de fluxo de controle e de dados, que devem ser investigados, e a *Jaguar Viewer* que apresenta a informação visualmente para auxiliar o especialista em depuração. O trabalho reforça o motivo de não existirem muitos trabalhos de localização de defeitos baseados em fluxo de dados destacando o custo computacional de se obter os exercícios de *dua*'s.

A *Jaguar Runner* atribui propensão apenas a elementos que são "cobertos" pelas execuções dos casos de teste e utiliza de dez heurísticas para isso. Os valores de propensão a defeitos obtidos pelas heurísticas possuem diferentes intervalos, então a ferramenta faz o processo de normalização desses valores, colocando-os em um intervalo de 0 à 1. O resultado então é salvo em um arquivo *XML*, que contém os valores de suspeita de cada elemento. A *Jaguar Viewer* por sua vez, utiliza as informações contidas no arquivo descrito para informar visualmente, com diferentes cores, os locais mais prováveis de conterem o defeito.

Utilizando esta ferramenta, Ribeiro *et al.* [30] compara o desempenho do espectro de fluxo de dados em relação ao espectro de fluxo de controle, para a localização de defeitos. A investigação foi aplicada à eficácia e eficiência dos espectros em seis programas Java. Os resultados mostraram que o espectro de fluxo de dados, por prover informação mais detalhada sobre o defeito, guia os desenvolvedores a encontrarem mais defeitos investigando o intervalo 7 a 40 linhas de código. O custo de obter a lista de elementos de fluxo de dados é consideravelmente maior (de 94% até 245%), porém ainda viável para programas do mundo real, demandando de 22 segundos até 8:35 minutos nos programas analisados.

2.2.3 Métricas de avaliação para métodos de Localização de Defeitos

A literatura oferece diferentes métricas para avaliar métodos de localização de defeitos. Algumas, tal como a *Exam* [36], analisam os resultados de um ponto de visto relativo, ou seja, os números são impactados pelo tamanho do software. Outras, tal como *Accuracy*, consideram apenas um número n finito e geralmente pequeno para investigar na avaliação.

Algumas das métricas comumente usadas são apresentadas a seguir, e categorizadas em: *métricas de eficácia* e *métricas de empate*.

Métricas de Eficácia

Métricas de eficácia são utilizadas para avaliar e comparar técnicas com respeito à habilidade para a localização de defeitos. Permitem, por exemplo, abstrair evidências de que um novo método possui maior precisão ou é competitivo em relação aos existentes.

Exam. A métrica *Exam* representa a proporção de elementos investigados até encontrar o local do primeiro defeito; em outras palavras, é a porcentagem de comandos analisados que não estão necessariamente relacionados ao defeito. Essa métrica produz números em razão ao número de comandos, logo é altamente influenciada pelo tamanho do programa.

Accuracy. Diferentemente da *Exam*, a *Accuracy* traduz a aptidão para localizar defeitos em valores absolutos. Representa o número de defeitos localizados investigando um número fixo n de elementos no topo do ranking de elementos suspeitos [7]. Essa métrica pode ser percebida por um especialista de depuração ao inspecionar um número limitado de elementos de programa: indica quantos defeitos foram localizados ao investigar os n elementos mais suspeitos.

Wasted Effort. Assim como a *Accuracy*, a *Wasted Effort* é uma métrica de valores absolutos e também necessita de um número fixo n para ser aplicada. Ela computa o número de elementos investigados até localizar o local do primeiro defeito, mas limitando-se a inspecionar os n elementos do topo do ranking de elementos suspeitos [7]. Representa a quantidade de elementos "não defeituosos" inspecionados ao seguir o *ranking* sugerido. Valores menores indicam que pouco esforço foi gasto inspecionando elementos não defeituosos.

Apesar das três métricas apresentadas serem de avaliação de eficácia, elas não representam os mesmos interesses. *Exam* e *Accuracy* destacam a posição do local do defeito em um *ranking* de propensão a defeito, mas a partir dos pontos de vista relativo e absoluto, respectivamente. Uma técnica de localização de defeitos é avaliada de forma positiva segundo a *Exam*, se em diversas versões defeituosas, ela geralmente deixa o local defeituoso em uma posição não muito longe do topo do *ranking*. Enquanto que para a *Accuracy*, uma técnica seria melhor avaliada se para diversas versões defeituosas, deixasse a maioria dos locais defeituosos no topo do *ranking*, mesmo que para algumas versões a técnica nem consiga destacar o local defeituoso dos não defeituosos, deixando-os no fim do *ranking*.

Métricas de Empate Crítico

Além da eficácia, outro ponto de análise para técnicas de localização de defeitos é quanto a empates gerados ao compor o *ranking* de propensão a defeito. Nessa característica, a investigação se dá em torno da dependência de estratégias de desempate presente nas técnicas de localização de defeitos. Por exemplo, se diferentes comandos recebem o mesmo valor de propensão a defeito, é necessária a aplicação de uma decisão de desempate, tais como as estratégias de *melhor caso*, *pior caso*, e *caso médio*.

Xu *et al.* [39] definem empate como um conjunto de instruções, em que para cada uma das quais foi atribuída a mesma pontuação de suspeita, portanto compartilha a mesma posição no *ranking* de propensão a defeito. Assim, um comando defeituoso pode estar empatado com comandos não defeituosos. Nesse sentido, os autores definem empate crítico como um empate que contém um comando defeituoso, os comandos em um empate crítico são chamados *comandos empatados criticamente*.

Um empate crítico afeta o *ranking* de propensão a defeito, pois agrega incerteza no cálculo de uma pontuação de suspeita dos elementos defeituosos do programa. Então, quanto maior o número de empates que envolvem comandos defeituosos, mais difícil é estimar com precisão em que posição do ranking o comando defeituoso será localizado [39]. Assim, reduzir o número de empates críticos melhora a precisão ao localizar defeitos em software.

Critical Score. Um método de localização de defeitos deve, além de indicar com destaque o comando defeituoso, reduzir (ou evitar) empates críticos. Para medir a ocorrência desses empates críticos, Xu *et al.* definem a métrica *Critical Score* (*CScore*) [39], que mede a proporção de empates críticos gerados por uma técnica de localização de defeitos, em relação ao número total de elementos do código. Portanto, é pertinente que uma técnica tenha número reduzido em relação ao *CScore*.

Com o *Critical Score* podemos comparar, por exemplo, técnicas que possuam performance semelhante quanto à eficácia para localização de defeitos, servindo como uma métrica complementar. Por não se tratar de uma métrica de eficácia, um valor ótimo de *CScore* não é suficiente para indicar a qualidade de uma técnica para encontrar defeitos.

2.3 Computação Evolucionária

Um dos principais objetivos da computação é automatizar solucionadores de problemas (*solvers*). Para isso, muitas vezes buscamos inspiração em solucionadores encontrados na natureza, como por exemplo o processo de evolução. A evolução das espécies apresentada por Darwin, pode ser utilizada como metáfora para a evolução de soluções em problemas de modelagem e otimização [12].

Considere um ambiente da natureza com diversos indivíduos da mesma espécie. O que garante que alguns indivíduos consigam sobreviver, o bastante para passar seus atributos para as próximas gerações é a aptidão de cada um, ou melhor, o quanto cada indivíduo está adaptado para aquele ambiente. No contexto da computação, tal cenário pode ser interpretado como um problema que possui diversas soluções, e para cada uma delas é possível calcular a sua qualidade, ou no caso, aptidão. A qualidade de cada solução dita se ela será melhor explorada, ou descartada.

Os indivíduos podem se reproduzir e também sofrer mutações, gerando assim outros indivíduos. Da mesma forma, as soluções podem ser combinadas e modificadas para gerar novas soluções. Dessa forma podemos montar uma analogia entre o *processo de evolução* e a inspiração deste para solucionar problemas pela computação, a última comumente denominada *Computação Evolucionária*; a Tabela 2.4 apresenta essa relação.

Processo Evolutivo		Computação Evolucionária
Ambiente	$\longleftrightarrow$	Problema
Indivíduo	$\longleftrightarrow$	Solução
Cromossomo	$\longleftrightarrow$	Representação da solução
Reprodução sexuada	$\longleftrightarrow$	Operador de cruzamento
Mutação	$\longleftrightarrow$	Operador de mutação
População	$\longleftrightarrow$	Conjunto de soluções
Gerações	$\longleftrightarrow$	Ciclos de execução

Tabela 2.4: Analogia entre o Processo Evolutivo e a Computação Evolucionária.

A *Computação Evolucionária* tem sido aplicada a diversos contextos, servindo principalmente de ferramenta para a solução de problemas de otimização, e também problemas de modelagem.

Um *Problema de Busca* pode ser representado como um sistema onde existem entrada (*input*), que aplicadas em um modelo, resultam em saída (*output*) como apresentado na Figura 2.4. O *Problema de Otimização* e o *Problema de Modelagem* se diferenciam principalmente em qual elemento é desconhecido e deve ser encontrado: entrada e modelo, respectivamente.

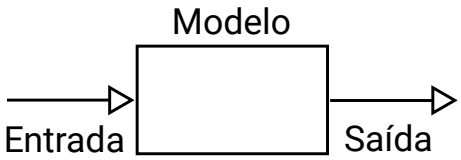

Figura 2.4: Problemas de Busca contém três elementos principais: entrada, modelo e saída.

Problemas de otimização: Em um problema de otimização o modelo é conhecido, e a saída é bem especificada, mas não é conhecido a entrada que implique na melhor saída (ou 'boa o suficiente'). Um exemplo é o problema da mochila (*knapsack problem*): dado a especificação de vários itens com pesos e preços diferentes, e também a capacidade máxima da mochila, o objetivo é encontrar a combinação de itens que representem o maior valor monetário, sem ultrapassar o peso máximo que a mochila pode conter. Neste problema, as entradas são combinações de itens, o modelo é a forma de calcular o preço e o peso total gerado pelos itens da entrada, e a saída é o preço total propriamente dito. Sabemos que queremos o maior valor de saída, mas conjunto de itens que o representam é desconhecido. Como exemplificado na Figura 2.5.

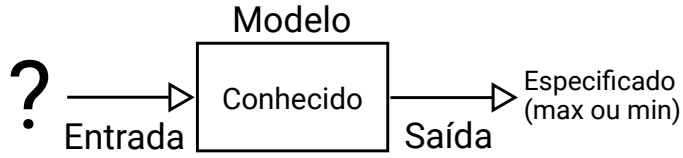

Figura 2.5: Problema de otimização: possui modelo conhecido e saída especificada, como a intenção de maximizar ou minimizar a saída, por exemplo. Porém a entrada que exerce essa intenção não é conhecida.

Problemas de modelagem: Diferentemente dos problemas de otimização, os problemas de modelagem não possuem conhecimento sobre o modelo, mas sim das entradas e saídas. O objetivo é encontrar um modelo que melhor relacione entradas com suas respectivas saídas, tal que o modelo encontrado para entradas e saídas conhecidas possa ser generalizado para futuros dados desconhecidos. Ou seja, através de um *processo de treino* (processo de aprendizagem) com dados conhecidos, buscamos encontrar um *modelo geral* para o problema.

Por exemplo, no contexto de moedas suponha que queremos descobrir como a valorização da Rúpia Indiana se relaciona com a valorização do Real. Através do histórico do último mês dos valores das duas moedas, intencionamos encontrar um modelo que melhor traduza a relação entre os dados de entrada (valorização da Rúpia Indiana) e os dados de saída (valorização do Real). Assim, temos entradas e saídas conhecidas, mas um modelo até o momento desconhecido. Esse modelo poderá ser usado nos próximos meses, para predizer a valorização do real se já tivermos a valorização da Rúpia Indiana. Vale ressaltar que a qualidade do novo modelo a ser encontrado está "intimamente" relacionada com a qualidade dos dados de treinamento. A Figura 2.6 ilustra problemas de modelagem.

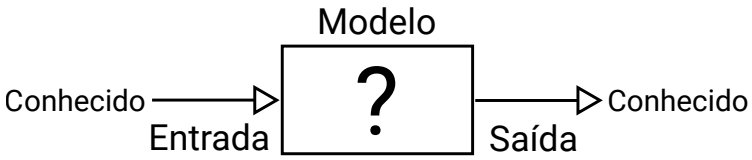

Figura 2.6: Problema de modelagem: possui entradas e saídas conhecidas, porém o modelo que as mapeia não é conhecido.

2.3.1 Algoritmos evolucionário

A computação evolucionária é concretizada comumente por algoritmos que traduzem as operações pertinentes a processos ligados à natureza. Tais algoritmos são, em geral, denominados *Meta-heurísticas Evolucionárias*. Gendreau e Potvin [14] definem "meta-heurísticas são métodos de solução que orquestram uma interação entre procedimentos de melhoria local e estratégias de nível superior para criar um processo capaz de escapar de ótimos locais e realizar uma pesquisa robusta de um espaço de solução".

No contexto de problemas de busca, os termos *procedimentos de melhoria local*, *estratégias de nível superior* e *processo capaz de escapar de ótimos locais* são relacionadas à busca por "soluções de qualidade"; *espaço de solução* se refere ao conjunto das possíveis soluções aos problemas de otimização e modelagem.

A definição de *meta-heurísticas*, conforme apresentada por Gendreau e Potvin [14], é melhor entendida nas subseções seguintes, que exploram duas meta-heurísticas evolucionárias pertinentes ao presente trabalho de pesquisa.

2.3.2 Algoritmo genético

Algoritmo Genético (GA – *Genetic Algorithm*), descrito pela primeira vez por John Holland em 1975 [16], é o membro mais conhecido do conjunto de algoritmos evolucionários [12], sendo principalmente utilizado para problemas de otimização, ou seja, a busca tem como objetivo encontrar entradas que produzam boas saídas. O GA segue um fluxo básico ilustrado na Figura 2.7.

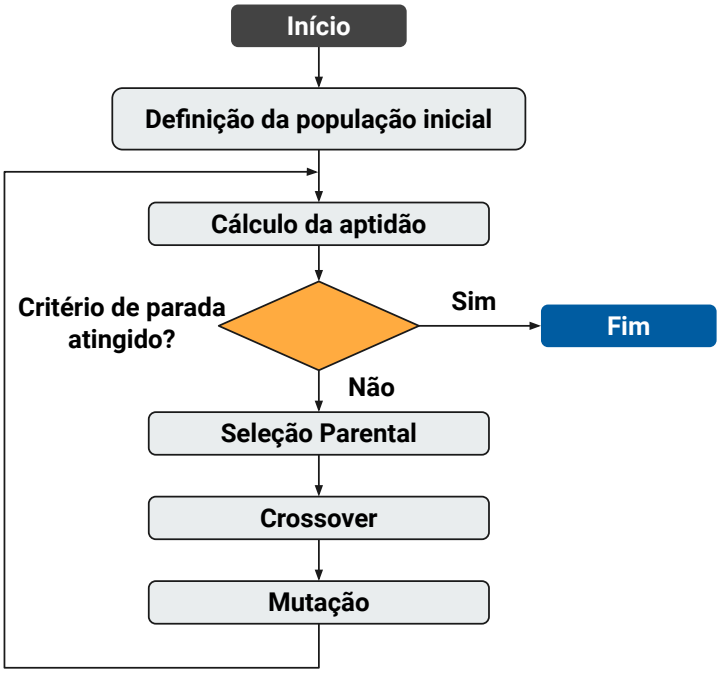

Figura 2.7: Fluxo básico de Algoritmo Genético [14]

No primeiro passo, ocorre a *definição da população inicial*, ou seja, são definidos os indivíduos que serviram de base para a exploração, geralmente estes primeiros indivíduos são gerados aleatoriamente, mas se há conhecimento prévio sobre o problema, soluções conhecidas podem compor a população inicial. No próximo passo, tem lugar o *cálculo da aptidão* (*fitness*) para cada indivíduo: refere-se a um valor que denota o quão

adaptado o indivíduo está. Esse valor será necessário para comparar a qualidade de diferentes de soluções para o problema em questão.

Ao avaliar cada solução, objetiva-se responder a questão: *critério de parada atingido?* Se for alcançado o critério de parada, então a execução é finalizada. Do contrário, segue para as etapas subsequentes: *seleção parental*, *recombinação* (*crossover*), e *mutação*. Ao fim dessas etapas, o ciclo do algoritmo é reiniciado, dando início a uma nova geração de soluções.

Seleção Parental

A seleção parental é a etapa responsável por selecionar, dentre os indivíduos da geração atual, aqueles que produzirão novos indivíduos por recombinação ou por mutação. Os métodos de seleção parental mais populares são o método da roleta e o método de torneio.

No *método da roleta*, cada indivíduo possui a probabilidade de ser escolhido proporcional ao seu valor de aptidão. Como podemos ver no exemplo da Figura 2.8, ao somar os valores de aptidão de todos os indivíduos, o valor obtido por *A* foi o maior, correspondendo à 61,6% do total, que será a probabilidade de ele ser escolhido em cada execução do método da roleta.

Indivíduo	Cadeia	x	Aptidão (x^2)
A	100100	36	1296
B	010010	18	324
C	010110	22	484
D	000001	1	1

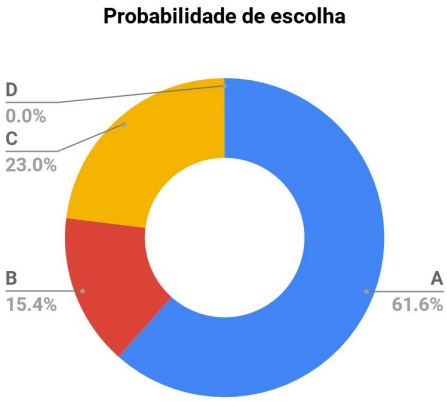

Figura 2.8: Exemplo de aplicação da roleta como método de seleção parental.

Apesar de os indivíduos mais aptos possuírem maior chance de serem selecionados, esse método também permite que indivíduos menos aptos sejam escolhidos. Tal é recomendável para aplicar uma pressão seletiva que priorize os bons indivíduos, mas não descarte os demais [14].

No **método de torneio**, um número k de indivíduos da população atual são sorteados, e aquele com a maior aptidão dentre os k indivíduos é selecionado e passa a compor o conjunto daqueles que serão utilizados na recombinação. Esse método possui

a vantagem de não exigir nenhum conhecimento sobre o tamanho da população, ainda permitindo ser usado quando a aptidão não é, necessariamente, um valor quantificável.

Recombinação (*Crossover*)

A recombinação é a operação de mescla do material genotípico de diferentes indivíduos. Essa operação age na busca propiciando a combinação de duas soluções com o objetivo de encontrar novas que sejam tão boas ou melhores do que as anteriores.

Alguns exemplos de estratégias de recombinação simples são: recombinação de um único ponto e recombinação de n-pontos [12]. A Figura 2.9 exemplifica a combinação de indivíduos usando *crossover* de um e dois pontos: pontos de corte são definidos aleatoriamente, os novos indivíduos serão gerados ao unir as partes dos diferentes pais.

A taxa de recombinação é a probabilidade de ocorrer ou não o *crossover* para algum indivíduo, então é possível que um indivíduo não participe de recombinação em alguma geração.

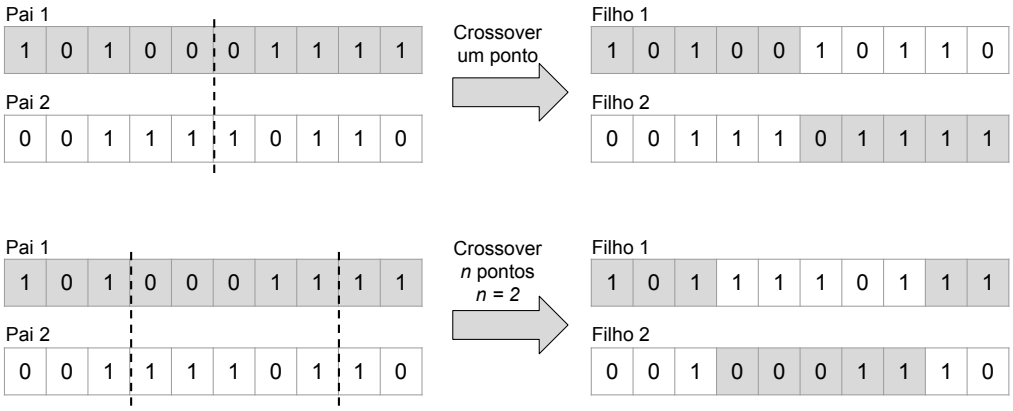

Figura 2.9: Exemplos de recombinação de um e dois pontos.

Diversas outras estratégias são apresentadas na literatura, como por exemplo a combinação uniforme e a baseada em ordem [14]. Porém, o que se destaca é a característica inerente aos operadores de recombinação de sempre gerar indivíduos “filhos” a partir de informações presentes nos indivíduos “pais” .

Mutação

Diferentemente da recombinação, a mutação é um operador de variação unário, ou seja, é aplicado a um único indivíduo. Nesse operador, o indivíduo em questão recebe uma pequena “perturbação genética”, resultando em um mutante (o indivíduo geneticamente modificado).

Em uma representação binária de soluções, a mutação pode ser simplesmente a variação de bits (*bitflip*), mas diferentes problemas podem demandar diferentes perturba-

ções nos indivíduos. A Figura 2.10 ilustra a mutação de *bitflip*, nela para cada posição do genótipo do indivíduo existe uma certa probabilidade de ser o aquele valor ser alterado, essa probabilidade é chamada de taxa de mutação [12]. Ao fim dessa operação, um novo indivíduo é produzido, trazendo possíveis soluções com diferentes atributos para a busca.

Figura 2.10: Exemplo de mutação por *bitflip*.

2.3.3 Programação genética

Em computação evolucionária, *Programação Genética* (GP - *Genetic Programming*) [21] é conhecida principalmente por ser aplicada em problemas de modelagem. Diferentemente do GA, cada indivíduo na GP é comumente representado por árvores [12], e os operadores de variação não são aplicados em sequência, como podemos ver na Figura 2.11. A cada geração de um novo indivíduo, a partir de outros já existentes, apenas um dos três operadores (*crossover*, mutação e reprodução) é aplicado.

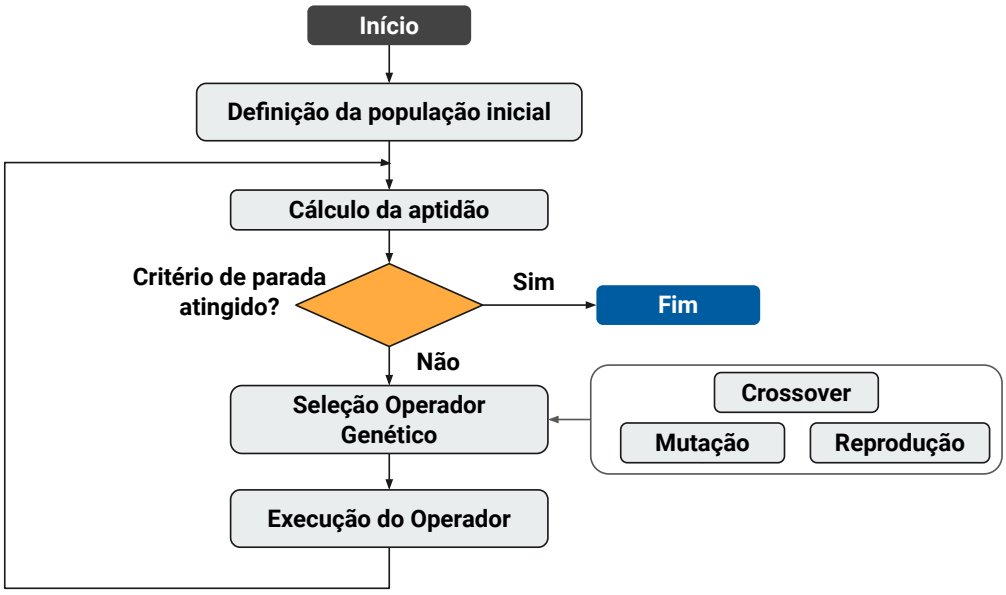

Figura 2.11: Fluxograma simplificado da Programação Genética, fluxo completo pode ser acessado em: www.genetic-programming.com/gpflowchart.html

Suponha um problema de regressão a partir de dados de entrada e saída conhecidos: uma possível solução (indivíduo) é uma equação, que pode ser representada por uma árvore. As equações existentes (indivíduos da população) podem ser recombinadas e alteradas para gerar novas equações. Para ilustrar, considere a operação ***crossover*** aplicada

em duas árvores (Pai 1 e Pai 2) na Figura 2.12; como resultado temos um único novo indivíduo (Filho). Por outro lado, a **reprodução** é uma operação onde o indivíduo é apenas copiado para a próxima geração, sem qualquer mudança em seu código genético.

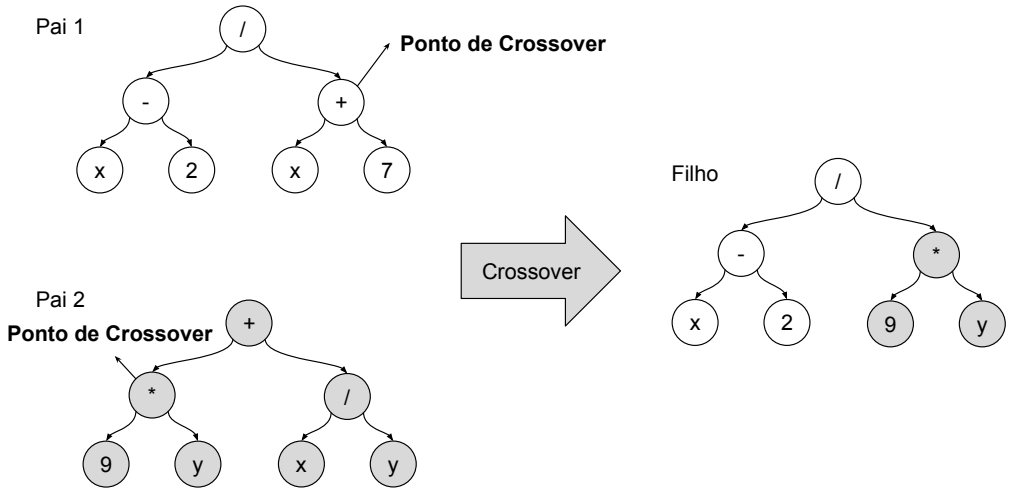

Figura 2.12: Exemplo de crossover em indivíduos de representação em árvore utilizado pela GP.

Existem variadas maneiras apresentadas na literatura para aplicar a operação **mutação** em indivíduos da GP, mas as principais e mais simples são a *mutação de um único ponto* e a *mutação de subárvore* [14]. A primeira é usada para aplicar uma perturbação leve aos indivíduos, já a segunda aplica uma mudança maior, podendo gerar uma solução mutante cujo código genético se distancia muito da solução original. A Figura 2.13 apresenta ambas as variações. Na mutação de um único ponto, escolhe-se um nó aleatório da arvore para ser o ponto de mutação, em seguida o valor do ponto de mutação é alterado por outro valor compatível. Na mutação de subárvore, um novo ramo é gerado aleatoriamente para substituir o ponto de mutação.

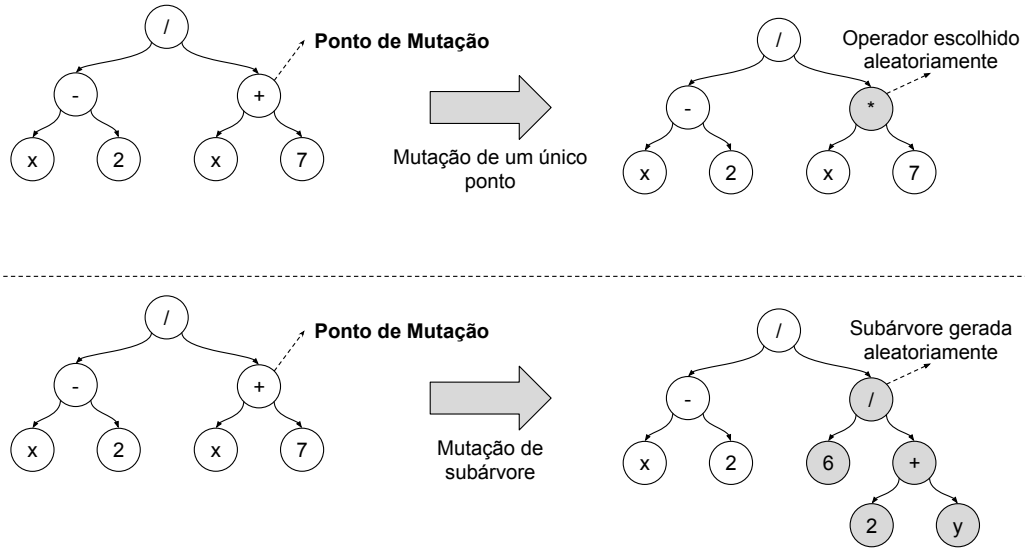

Figura 2.13: Exemplos de *mutação de um único ponto* e *mutação de subárvore*, ambas aplicadas a um mesmo indivíduo de GP.

Função Aptidão

Por ser aplicada em problemas de modelagem, uma função aptidão (*fitness*) na GP é, por conceito, distinta daquelas aplicadas ao GA. Uma função aptidão na GP refere-se essencialmente a medições de indivíduos, tais como para calcular: o *erro* entre o *output* gerado pela solução candidata e o *output* esperado; a *acurácia* do modelo gerado para reconhecer padrões.

Em geral, é necessário dividir os dados de entrada conhecidos em dois conjuntos: *conjunto de treino* e *conjunto de validação*. Todo o conjunto de treino é executado para cada solução candidata (modelo) gerada durante o processo de busca da GP [14]. Esse aspecto faz com que a GP seja por muitas vezes considerada muito custosa para ser utilizada em alguns problemas, já que são encontrados diversas soluções candidatas no decorrer das gerações.

2.4 Localização de Defeitos Baseada em Busca

Engenharia de Software Baseada em Busca (SBSE - *Search Based Software Engineering*) lida com problemas de Engenharia de Software, tratando-os como problemas de busca – problemas de otimização e modelagem – que, por sua vez, podem ser abordados por diversas estratégias de busca conhecidas, como métodos evolucionários e de busca local [15].

Harman *et al.* [15] destacam dois aspectos essenciais ao se aplicar estratégias de busca a problemas de Engenharia de Software: a escolha da representação do problema e

a definição da função de avaliação de possíveis soluções (*fitness*).

Wang *et al.* [32] utilizam SBSE de forma seminal para o problema de localização de defeitos, introduzindo assim a denominação Localização de Defeitos Baseada em Busca (SBFL - *Search-Based Fault Localization*). Este trabalho representa cada solução como um conjunto de pesos, que devem ser aplicados às heurísticas tradicionais, gerando assim uma combinação linear dos valores de propensão a defeito. A Equação 3-4 no Capítulo 3 ilustra a presença do conjunto de pesos para a combinação de heurísticas. Para avaliar as soluções encontradas pelos métodos de busca, Wang *et al.* [32] empregam como função *fitness* o valor médio da métrica *Exam*, que é descrita na Subseção 2.2.3, com o objetivo de alcançar a minimização dos valores de *fitness* de cada possível solução.

Em estudo recente (2020) de revisão de literatura Leitao-Junior *et al.* [22] caracterizam os métodos propostos no contexto de SBFL, identificando fontes de informações sobre defeitos, funções de avaliação, algoritmos (meta-heurísticas) e elementos referentes aos experimentos descritos, tais com *benchmarks*, *baselines*, e métricas de avaliação.

Pode-se identificar, dentre as oportunidades de pesquisa elencadas em Leitao-Junior *et al.*, que há uma lacuna para explorar diversas fontes de defeitos e uma carência de pesquisas que empreguem informação de fluxo de dados em conjunto com meta-heurísticas evolucionárias. Tal combinação é o objeto do presente trabalho de pesquisa, na busca por melhores abordagens de localização de defeitos em software.

2.5 Considerações Finais

O presente capítulo apresentou as informações que fundamentam o entendimento da pesquisa. Foram explorados conteúdos tais como: teste de software, especialmente sobre elementos estruturais cobertos durante o teste; espectros de fluxo de controle e de dados, que são considerados fonte de informação sobre potenciais defeitos; métricas de avaliação para métodos de localização de defeitos; problemas de busca; computação evolucionária e algumas de suas meta-heurísticas, na forma de uma ferramenta que apoia a modelagem e a solução de problemas de busca; e a localização de defeitos em si tratada como um problema de busca.

A localização de defeitos baseada em busca (SBFL – *Search-Based Fault Localization*) é um campo de pesquisa que aplica o paradigma SBSE (S*earch-Based Software Engineering*) ao problema de localização de defeitos. O estado da arte da SBFL – contexto da presente pesquisa – é apresentado em estudo recente [22], onde se identificou diversas oportunidades de pesquisa.

Essencialmente, na SBFL um algoritmo explora o espaço de busca, de modo que cada elemento desse espaço denote uma solução candidata relacionada a um local potencialmente defeituoso [22]. Os próximos capítulos introduzem: as abordagens propostas

pela presente pesquisa (Capítulo 3); e a avaliação conduzida para investigar a eficácia dessas abordagens (Capítulo 4).

CAPÍTULO 3

Abordagens Propostas

Este capítulo apresenta duas abordagens evolucionárias no contexto de *localização de defeitos baseada em fluxo de dados*, utilizando as meta-heurísticas *Algoritmo Genético* e *Programação Genética*, com foco em composição de heurísticas existentes e em geração de novas heurísticas, respectivamente. O primeiro teve seu conteúdo publicado e apresentado no *The 35th ACM/SIGAPP Symposium On Applied Computing* em artigo intitulado *Data-Flow-Based Evolutionary Fault Localization* [33], conforme descrito nas Seções 3.2 e 3.4. O último será explorado em artigo próximo, a ser concluído após a defesa do presente trabalho de mestrado.

O contexto e conclusões do capítulo anterior (Capítulo 2) são mostrados na Seção 3.1. A Seção 3.2 propõe a utilização de informação de fluxo de dados pela combinação evolucionária de heurísticas tradicionais de localização de defeitos: a Subseção 3.2.1 formaliza três medidas de propensão a defeitos, denominadas *propensão node*, *propensão dua* e *propensão dua-to-node*, em que as duas últimas são baseadas em informação fluxo de dados; a Subseção 3.2.2 ilustra a aplicação das três propensões da seção anterior, por meio de um exemplo motivacional; a Subseção 3.2.3 introduz uma abordagem para a localização de defeitos, baseada em *Algoritmo Genético*, pela combinação de heurísticas existentes e pelo emprego de informação de fluxo de dados.

A Seção 3.3 propõe a conversão das informações de fluxo de dados em oito variáveis, as quais serão usadas para gerar novas equações vocacionadas à localização de defeitos: a Subseção 3.3.1 formaliza as oito variáveis baseadas de fluxo de dados; a Subseção 3.3.2 apresenta um exemplo de aplicação. A Seção 3.3.3 introduz uma abordagem evolucionária pelo emprego do algoritmo *Programação Genética* e uso das novas variáveis formalizadas, visando à geração de novas heurísticas de localização de defeitos.

A Seção 3.4 define uma nova métrica de avaliação, que é destinada a investigar o número de empates críticos presentes nos *rankings* de propensão a defeitos. Tal métrica é pertinente para investigar o quão dependente de estratégias de desempate as técnicas de localização de defeitos são.

A Seção 3.5 apresenta um breve debate para a conclusão do capítulo. Na Seção 3.6 há a indicação de desenvolvimentos futuros às abordagens postas.

3.1 Contexto

Localização de defeitos é um estágio essencial na depuração de programas. Conhecida por demandar bastante tempo e dinheiro, é também considerada extremamente monótona. A automatização dessa prática pode resultar em impacto importante na produtividade de desenvolvedores e na confiabilidade de software.

Nesse contexto podemos destacar algumas conclusões do Capítulo 2:

- Os métodos de localização de defeitos baseados em espectro de cobertura se mostram promissores. Aplicando heurísticas sobre os dados de cobertura obtidos pela execução de casos de teste, é possível computar uma propensão a defeito para cada elemento do código investigado. Porém estudos mostram que há muito o que se evoluir [38].
- A análise de fluxo de dados tem sido utilizada em métodos para a localização de defeitos, obtendo bons resultados quando comparada à análise de cobertura de nós [24]. Estudos que utilizam a análise de fluxo de dados [24, 31] apontam que diferentes tipos de defeitos são melhor localizados por elementos de natureza distinta no espectro de cobertura, tais como cobertura de nós, cobertura de arcos e cobertura de associações de fluxo de dados. Assim, métodos que utilizam diferentes elementos podem ser combinados em busca de uma solução mais geral.
- Técnicas de localização de defeitos baseadas em busca [32, 40] tratam o problema em questão como um desafio de otimização: utilizam algoritmos de busca para gerar novas heurísticas e para combinar diferentes heurísticas existentes. Guiadas por métricas de localização de defeitos, estes métodos mostram que há diferentes formas de combinar informação de distintas naturezas, para se alcançar resultados mais efetivos.

3.2 Abordagem GA - Combinação de heurísticas para localização de defeitos baseada em fluxo de dados

Esta abordagem consiste em combinar diferentes heurísticas, pelo emprego da meta-heurística *Algoritmo Genético* assim como apresentado por Wang *et al.* [32], mas usando informação proveniente de ambos os fluxos: fluxo de controle e fluxo de dados. Para um melhor compreendimento do método, se mostra necessária a definição de alguns termos.

3.2.1 Definições sobre localização de defeitos baseada em fluxo de dados

Assim como a cobertura de nós pode ser utilizada para a localização de defeitos, a cobertura de associações de fluxo de dados é um meio alternativo para capturar *pistas* sobre defeitos existentes. A análise de cobertura de associações de fluxo de dados (*Definition-Use Associations – dua's*, ou simplesmente *duas*) agrega potencial informação de possíveis variáveis envolvidas no local do defeito, o que caracteriza um recurso adicional à análise de cobertura de fluxo de controle.

Para elencar a posição dos elementos defeituosos no ranking usando informação de fluxo de dados, Santelices *et al.* [31] e Ribeiro *et al.* [30] utilizam uma abordagem similar, colocando todos os comandos que participam de uma dua em uma mesma **posição do ranking**.

A abordagem apresentada por Wang *et al.* [32] utiliza o valor de propensão a defeitos de cada comando obtido através de diferentes heurísticas. Ao aplicar uma abordagem similar, precisamos então converter o valor de suspeita de elementos fluxo de dados, para cada comando que participa da dua em questão. Dessa forma os comandos ficarão com o maior valor de **propensão a defeito** de uma dua em que participam.

Na prática a estrategia utilizada é muito similar à de Santelices *et al.* e Ribeiro *et al.*, mas focando no valor de propensão à defeito e não na posição do ranking de elementos suspeitos. Assim, se mostra necessária a formalização desta estratégia.

Definição 1 - ***Propensão node***. Refere-se aos métodos tradicionais de localização de defeitos, tais como Tarantula [19] e Ochiai [1], que utilizam como base o espectro de cobertura de comandos (ou nós); ou seja, é a propensão de defeitos baseada na cobertura de fluxo de controle. As variáveis de cobertura obtidas pela execução dos casos de teste trazem a informação dos possíveis locais do defeito, mostrando para cada nó se ele foi executado (**e**, *i.e. executed*) ou não (**n**, *i.e. non-executed*) e se o caso de teste que o executou é positivo (**s**, *i.e. successful*) ou negativo (**f**, *i.e. failed*). Dessa forma, as variáveis são rotuladas como: $ef(node)$, $es(node)$, $nf(node)$ e $ns(node)$, em que $node$ refere-se ao nó em questão. Essas variáveis são utilizadas na formalização de heurísticas para calcular o valor de propensão a defeito que um comando possui.

Definição 2 - ***Propensão dua***. De forma análoga à *propensão node*, a análise de fluxo de dados deriva variáveis com respeito à cobertura de associações de fluxo de dados (cobertura de *duas*), a saber $ef(dua)$, $es(dua)$, $nf(dua)$ e $ns(dua)$. A *propensão dua* é a aplicação dessas variáveis às fórmulas das heurísticas originalmente definidas para a cobertura de comandos (ou nós). Ou seja, a *propensão dua* são os métodos de suspeição a defeitos tradicionais aplicados à cobertura de associações de fluxo de dados na execução de casos de teste. A Equação 3-1 apresenta a heurística *Tarantula*, conforme definida na

Equação 2-1, mas aplicada a fluxo de dados, em que *dua* refere-se a uma das associações de fluxo de dados do programa.

$$Tarantula(dua) = \frac{\frac{ef(dua)}{ef(dua)+nf(dua)}}{\frac{ef(dua)}{ef(dua)+nf(dua)} + \frac{es(dua)}{es(dua)+ns(dua)}} \tag{3-1}$$

Definição 3 - **Conjunto *alldua(node)***. Cada nó do programa (*node*) pode estar relacionado a várias associações de fluxo de dados (*duas*), conforme ressaltado no Capítulo 2. O conjunto $alldua(node)$ (Equação 3-2) contém todas as *duas* em que o nó investigado está presente.

$$alldua(node) = \bigcup dua_i \quad | \, node \in dua_i \tag{3-2}$$

Definição 4 - ***Propensão dua-to-node***. Nos métodos tradicionais baseados em fluxo de controle, um elemento de programa investigado está relacionado a um único nó (comando). No caso da *propensão dua*, a medida de suspeição é computada para uma associação de fluxo de dados, que envolve os vários nós (ou comandos) que pertencem a associação; isto é, a medida de suspeição de uma *dua* é compartilhada pelos nós que pertencem à *dua*. Controversamente, um mesmo nó pode participar de várias *duas*, podendo ainda haver valores distintos de suspeição em cada dessas *duas*. Portanto, a *propensão dua* não é trivialmente comparável com a *propensão node*. É preciso ponderar os valores de propensão *dua* para os respectivos nós que a compõem.

Incorporando informações de fluxo de dados às métricas de localização de defeitos baseadas em espectro de cobertura, a *propensão dua-to-node* representa a propensão a defeito relativa a um nó (ou comando), obtida a partir dos valores de *propensão dua* pertinentes às *duas* que tal nó participa. A Equação 3-3 representa a *propensão dua-to-node* ($\widehat{H}(node)$) para qualquer heurística H. A *propensão dua-to-node* é definida pelo maior valor de propensão $H(dua)$ dentre todas as *duas* que o nó investigado (*node*) faz parte.

$$\widehat{H}(node) = max\big(H(dua_i)\big) \quad | \, dua_i \in alldua(node) \tag{3-3}$$

Para verificar a aplicabilidade da *propensão dua-to-node*, um estudo piloto foi aplicado utilizando o programa *Tcas*, do conjunto de programas *Siemens Suite* [17], que é um *benchmark* muito empregado em pesquisas de localização de defeitos [41]. Foram utilizadas 35 versões defeituosas do programa, cada versão possuindo um único defeito. Neste estudo, foram calculadas a *propensão node* e a *propensão dua-to-node* usando 11 heurísticas.

Como métrica de avaliação de resultados, foi utilizada *Exam* [36], que pode ser interpretada como a posição do comando defeituoso em um *ranking* de propensão a defeito, relativo à quantidade de comandos do código investigado. A Figura 3.1 traz a avaliação das propensões quanto à métrica *AvgExam* – média dos valores de *Exam* considerando as 35 versões defeituosas utilizadas do programa, e aplicando 11 heurísticas de localização de defeitos, tais como *Tarantula*, *Ochiai* e *Zoltar*.

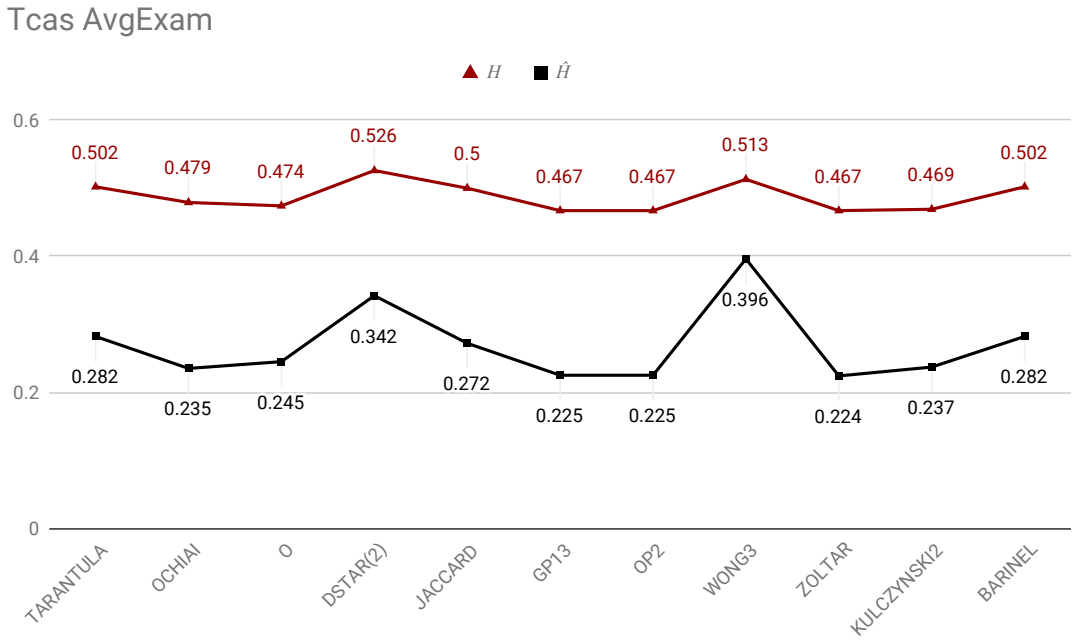

Figura 3.1: Experimento piloto no programa Tcas.

Para o *Tcas*, a propensão *dua-to-node* ($\widehat{H}(node)$) apresenta resultados competitivos para todas as heurísticas analisadas, com valores até $0,24$ menores que os obtidos pelo método tradicional baseado em fluxo de controle. Um exemplo é a heurística Ochiai [1], cujo resultado para a propensão *dua-to-node* foi aproximadamente 50% melhor do que a Ochiai *node*.

3.2.2 Exemplo de aplicação

Tabela 3.1: Exemplo de localização de defeitos baseada em cobertura de associações de fluxo de dados.

dua	tc1 (1,1,3)	tc2 (1,2,3)	tc3 (3,2,1)	tc4 (3,3,3)	tc5 (3,1,2)	tc6 (2,1,3)	$Tarantula(dua)$
(m, 1, 19)				•	•		**0.83**
(x, 0, 3-6)	•				•	•	0.71
(y, 0, 3-6)	•				•	•	0.71
(y, 0, 2-3)	•	•			•	•	0.62
(z, 0, 2-3)	•	•			•	•	0.62
(x, 0, 1)	•	•	•	•	•	•	0.5
(m, 4, 19)		•					0
(m, 7, 19)	•					•	0
(m, 12, 19)			•				0
(x, 0, 3-4)		•					0
(x, 0, 6-7)	•					•	0
(x, 0, 7)	•					•	0
(x, 0, 11-12)			•				0
(x, 0, 11-14)				•			0
(y, 0, 2-11)			•	•			0
(y, 0, 4)		•					0
(y, 0, 11-12)			•				0
(y, 0, 11-14)				•			0
(y, 0, 12)			•				0
Sucesso / Falha	S	S	S	S	F	S	

A Tabela 3.1 ilustra um exemplo de localização de defeitos baseada em cobertura de associações de fluxo de dados. A tabela mostra que o comando defeituoso (Comando 1) está presente em mais de uma *dua*, por exemplo em $< m, 1, 19 >$ e $< x, 0, 1 >$, cada dessas *duas* possui uma medida própria de propensão. Para calcular a propensão do Comando 1, já que esse comando participa de várias *duas*, utilizamos a *propensão dua-to-node*: $\widehat{H}(node)$. A Tabela 3.2 traz uma comparação entre as propensões obtidas por *Tarantula* e $\widehat{Tarantula}$ para cada linha do programa exemplo apresentado na Tabela 2.2. É possível perceber que a *propensão dua-to-node* atribui maior valor de suspeita ao comando defeituoso. Porém, é importante observar o número total de elementos empatados (que possuem o mesmo valor de propensão): o número de empates em $\widehat{Tarantula}$ é superior relativo ao *Tarantula* tradicional. Nesse sentido, a Seção 3.4 introduz uma métrica para lidar como o cenário *empates críticos*.

Tabela 3.2: Comparação entre *Tarantula* e $\widehat{Tarantula}$, a Linha 1 representa o local exato do defeito.

Linha	*Tarantula*	$\widehat{Tarantula}$
0	0.5	0.71
1	0.5	**0.83**
2	0.5	0.62
3	0.62	0.71
4	0	0
5	0	0
6	**0.71**	0.71
7	0	0
8	0	0
9	0	0
10	0	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0	0
18	0	0
19	0.5	**0.83**
20	0	0

3.2.3 Composição de heurísticas para a localização de defeitos

Wang *et al.* [32] exploram a composição de heurísticas usando informações fluxo de controle através da meta-heurística *Algoritmo Genético*, modelando o problema de localização de defeitos como um problema de otimização. A modelagem encontra combinações lineares de n ($n = 22$) heurísticas. A estratégia utilizada é a busca por pesos w_k para cada heurística H_k $(1 <= k <= n)$. Dessa forma, o método tem como solução a soma ponderada HC dos valores de propensão a defeito obtidos pelas 22 heurísticas, como descrito na Equação 3-4.

$$HC(node) = w_1 \times H_1(node) + w_2 \times H_2(node) + ... + w_n \times H_n(node) \qquad (3\text{-}4)$$

Wang *et al.* [32] representam as soluções como uma cadeia de 7 bits para cada peso w_k, de forma que $0 \leq w_k \leq 1$. A qualidade de cada solução é medida pela média da métrica *Exam* (a posição do comando defeituoso no *ranking* de maiores propensões), relativa à quantidade de versões defeituosas.

De forma análoga, o presente trabalho investiga combinações lineares de heurísticas, mas baseadas em fluxo de dados. Para cada uma das n heurísticas, é calculado

$H(node)$ e $\widehat{H}(node)$. Assim, podemos gerar três composições: uma combinação simples (HC) de propensões *node* que usa apenas dados de fluxo de controle (Equação 3-4); uma combinação simples de propensões *dua-to-node* $\left(\widehat{HC}\right)$ utilizando apenas informações de fluxo de dados (Equação 3-5); e uma combinação híbrida que combina informação de fluxo de controle e fluxo de dados $\left(HC_{hyb}\right)$, como mostrado na Equação 3-6.

$$\widehat{HC}(node) = w_1 \times \widehat{H}_1(node) + w_2 \times \widehat{H}_2(node) + ... + w_n \times \widehat{H}_n(node) \tag{3-5}$$

$$\begin{aligned} HC_{hyb}(node) = {} & w_1 \times H_1(node) + w_2 \times H_2(node) + ... + w_n \times H_n(node) + \\ & w_{n+1} \times \widehat{H}_1(node) + w_{n+2} \times \widehat{H}_2(node) + ... + w_{n+n} \times \widehat{H}_n(node) \end{aligned} \tag{3-6}$$

O Capítulo 4 analisa empiricamente as combinações lineares HC, $\widehat{HC}$ e HC_{hyb}, pelo uso de várias métricas de avaliação usadas na literatura.

3.3 Abordagem GP - Criação de heurísticas para a localização de defeitos baseada em fluxo de dados

Esta abordagem consiste na criação de novas equações que utilizem variáveis de cobertura obtidas tanto da análise de fluxo de controle, quanto da análise de fluxo de dados. Para isso é preciso computar valores de cobertura para cada elemento de programa, a partir da cobertura de fluxo de dados (*duas*). A seguir são introduzidas novas variáveis de cobertura, cujos valores distinguem elementos de programa em que ocorrem definição de dados e uso de dados, durante a análise das *duas* exercitadas pelos casos de teste.

3.3.1 Definição das novas variáveis de cobertura para fluxo de dados

Heurísticas para Localização de defeitos possibilitam a criação de um *ranking* dos comandos mais propensos ao defeito, pela atribuição de um valor de suspeita para cada comando. Como mostrado na Seção 2.1.1, uma *dua* pode ter dois, ou três nós (comandos) associados (por exemplo:<a,3,4> e <a,0,1–2>). Ao calcular o valor de propensão a defeito, as heurísticas atribuem um mesmo valor de suspeita para cada um dos nós (comandos) pertencentes à mesma *dua*. Entretanto, esse comportamento prejudica a técnica de localização de defeitos, visto que a torna menos acurada, pois força diversos comandos (nós) a compartilharem o mesmo valor de propensão a defeito.

Na presente abordagem – *Abordagem GP* – a meta-heurística Programação Genética é aplicada para gerar novas equações usando informação de fluxo de dados.

Para cada associação de fluxo de dados (*dua*), é feita a distinção entre a ocorrência de definição de dados (*comando def*) e a ocorrência de uso de dados (comando uso). Dessa forma, seria possível calcular, separadamente, a propensão a defeito para o *comando def* e a propensão a defeito para o *comando uso*. Como consequência, introduzimos oito novas variáveis de cobertura derivadas da análise de fluxo de dados. Elas são divididas em dois conjuntos:

- *local de definição* ($es_{def}(node)$, $ef_{def}(node)$, $ns_{def}(node)$, $nf_{def}(node)$);
- *local de uso* ($es_{use}(node)$, $ef_{use}(node)$, $ns_{use}(node)$, $nf_{use}(node)$).

Variáveis de cobertura fluxo de dados As novas variáveis de cobertura para *local de uso* são: (i) $es_{use}(node)$, proporção de casos de teste que exercitam *duas* que incluem o comando sob observação em *uso*, e finalizam com sucesso; (ii) $ef_{use}(node)$, proporção de casos de teste que exercitam *duas* que incluem o comando sob observação em *uso*, e finalizam com falha; (iii) $ns_{use}(node)$, proporção de casos de teste que não exercitam *duas* que incluem o comando sob observação em *uso*, e finalizam com sucesso; (iv) $nf_{use}(node)$, proporção de casos de teste que não exercitam *duas* que incluem o elemento sob observação em *uso*, e finalizam com falha. De forma similar, é possível obter as quatro variáveis de cobertura para *local de definição*, a saber: $es_{def}(node)$, $ef_{def}(node)$, $ns_{def}(node)$, $nf_{def}(node)$.

O algoritmo para calcular o valor de cada variável de cobertura de fluxo de dados é apresentado pela função a seguir. A função pode ser aplicada a qualquer das oito variáveis descritas anteriormente, mas escolhemos a variável es_{use} como exemplo:

Para cada comando s, é possível obter o valor normalizado para os valores *es* de todas *duas* que contenham s, como a Equação 3-7 mostra.

- $D(s)$ representa o conjunto de todas *duas* em que s está contida como um *local de uso* de dados;
- i é o índice de cada *dua* em $D(s)$;
- n representa a cardinalidade do conjunto $D(s)$;
- es_i representa a variável de cobertura *es* de cada *dua*;
- tc representa o número total de casos de teste;

$$es_{use}(s) = \frac{\sum_{i=1}^{i=n} es_i}{(n * tc)} \tag{3-7}$$

3.3.2 Exemplo de aplicação

Para melhor esclarecer a abordagem, considere: Figura 3.2 possui um exemplo com ocorrências de definição e de uso de variáveis; Tabela 3.3 apresenta as quatro

variáveis de cobertura, descritas anteriormente no Capítulo 2, para a execução de 20 casos de teste (10 de sucesso e 10 de falha). A partir destes dados, é possível obter as novas variáveis de cobertura de fluxo de dados introduzidas na presente abordagem.

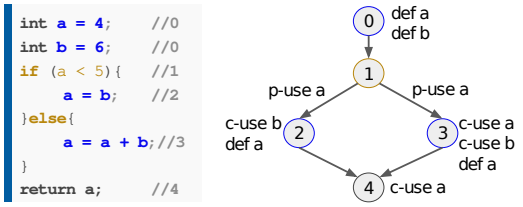

Figura 3.2: Exemplo motivacional.

dua			*es*	*ef*	*ns*	*nf*
var	def	use				
a	0	1 - 2	4	6	5	5
a	0	1 - 3	6	4	6	4
b	0	2	10	0	0	10
a	0	3	7	3	6	4
b	0	3	8	2	3	7
a	2	4	5	5	1	9
a	3	4	9	1	0	10

Tabela 3.3: Exemplo de cobertura de *duas*.

A Tabela 3.4 mostra os valores obtidos para as quatro novas variáveis referentes ao *local de uso*, de acordo com os dados presentes na Tabela 3.3. De forma análoga é possível obter as novas quatro variáveis referentes ao *local de definição*.

Perceba que no exemplo descrito, o nó (s) 1 participa em *local de uso* da *dua* <*a, 0, 1-2*> e da dua <*a, 0, 1-3*>, logo $n = 2$, a variável *es* possui valores 4 e 6 respectivamente para essas *duas*, como pode ser visto na Tabela 3.3. Assim ao aplicar a Equação 3-7, o somatório de *es* de todas as *duas* em que o nó 1 aparece em *local de uso* é 10. O número total de casos de teste *tc* é 20, logo temos $\frac{10}{2*20}$ para obter $es_{use}(1)$, resultando em 0.25.

Veja que para cada *dua* em que o Nó 1 participa em *local de uso*, há a possibilidade da mesma ser executada com sucesso (*es*) por todos os 20 casos de teste. Então a soma de *es* das *duas* seria no máximo 40. Mas elas de fato obtiveram a soma de *es* igual a 10, logo $es_{use}(1)$ possui o valor *0.25* referente a 25% em relação valor total de casos de teste, que possivelmente poderiam ser de sucesso ao executar as *duas* em que o nó 1 participa em *local de uso*.

s	n	es_{use}	ef_{use}	ns_{use}	nf_{use}
0	0	0	0	0	0
1	2	10/(2*20) = **0.25**	10/(2*20) = **0.25**	11/(2*20) = **0.275**	09/(2*20) = **0.225**
2	2	14/(2*20) = **0.35**	06/(2*20) = **0.15**	05/(2*20) = **0.125**	15/(2*20) = **0.375**
3	3	21/(3*20) = **0.35**	09/(3*20) = **0.15**	15/(3*20) = **0.250**	15/(3*20) = **0.250**
4	2	14/(2*20) = **0.35**	06/(2*20) = **0.15**	01/(2*20) = **0.025**	19/(2*20) = **0.475**

Tabela 3.4: Exemplo de valores pertinentes às variáveis *local de uso*.

3.3.3 Geração de heurísticas para a localização de defeitos

Yoo *et al.* [40] apresentam uma abordagem evolucionária para criar novas heurísticas aplicadas a localização de defeitos. As equações geradas por essa abordagem utilizam as quatro variáveis de cobertura de nós presentes em heurísticas tradicionais,

e conseguem se destacar quanto a eficácia. A *Programação Genética* apresentada por Yoo *et al.* explora a representação em árvore desse algoritmo, para a criação de equações (heurísticas) para lidar com o problema em questão.

O presente trabalho propõe a geração de equações que também considere as potencialidades das oito novas variáveis de fluxo de dados. Na Figura 3.3, 12 variáveis são mencionadas para a produção de novas heurísticas para a localização de defeitos: as quatro variáveis tradicionais de fluxo de controle e as oito variáveis descritas na Seção 3.3.1. A Figura 3.4 ilustra duas árvores que representam exemplos de equações com algumas das variáveis presentes na Figura 3.3.

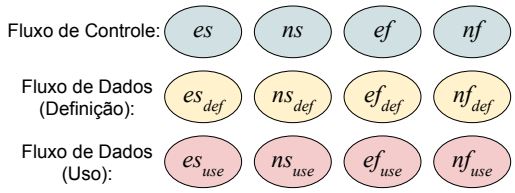

Figura 3.3: 12 variáveis de cobertura que podem ser utilizadas para criar novas equações.

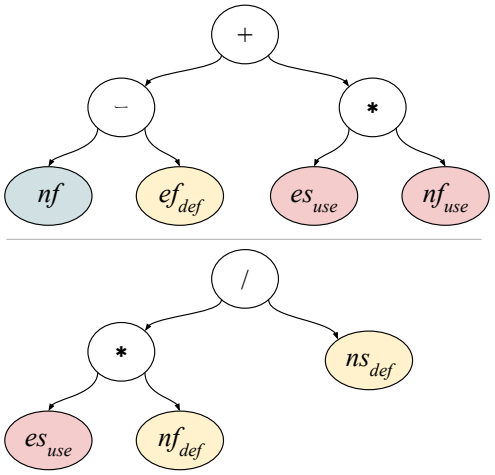

Figura 3.4: Exemplos de equações utilizando as novas variáveis.

Sumariamente, a abordagem proposta consiste em utilizar *Programação Genética*, pela extensão da abordagem de Yoo *et al.* [40] mas baseada em informação oriunda da análise de fluxo de dados, para construir equações a serem aplicadas como heurísticas para a localização de defeitos em software. Novas equações então podem ser produzidas, a partir das 12 variáveis de cobertura, o que inclui as variáveis propostas para fluxo de dados: quatro variáveis pertinentes a *local de uso* e quatro relativas a *local de definição*. Em adição, a introdução de constantes agrega maior diversidade às potencialidades das heurísticas produzidas.

É esperado que a abordagem proposta tenha maior acurácia para determinar os locais de defeitos em relação às heurísticas tradicionais, além de apontar tipos de defeito que de outra forma não seriam localizados. Uma investigação empírica nesse sentido é descrita no Capítulo 4.

3.4 Empate crítico absoluto

Como descrito anteriormente na Seção 2.2.3, Xu *et al.* [39] propuseram a métrica *CScore* (empate crítico), cujo valor é a *proporção de elementos de código* cuja medida de suspeita está empatada com o elemento defeituoso. Por se tratar de uma métrica de proporção, o valor computado sofre influência do tamanho do software: a análise do valor medido requer ponderar o número de linhas de código do software em estudo.

Porém, muitas métricas de eficácia utilizadas em localização de defeitos são formalizadas a partir de um ponto de vista absoluto; por exemplo, a computação da medida considera a análise de n elementos com maior valor de propensão a defeito. Em adição, o ponto de vista absoluto aproxima-se da perspectiva do testador do software, pois inspecionar n linhas de código é uma medida de esforço real em software de distintos tamanhos.

Dessa forma, a presente pesquisa estendeu a definição de de Xu *et al.* para um panorama de valor absoluto. Com o objetivo de contribuir para a análise de técnicas de localização de defeitos e investigar tais técnicas mais profundamente, introduzimos a métrica de *Empate Crítico Absoluto – actie@n* (*Absolute Critical Tie*).

A métrica *actie@n* computa o número de ocorrências de empate crítico nos n elementos do ranking (*top-n*) de maior valor de propensão a defeito: o número de elementos de programa não defeituosos que possuem o mesmo valor de suspeita que os elementos defeituosos e que estão entre os n primeiros elementos do *ranking* de suspeita. Representa o potencial esforço perdido com empates críticos por uma técnica até encontrar o defeito. Maiores valores nessa métrica indicam uma maior dependência de estratégias de desempate, e uma menor precisão ao localizar defeitos em software.

Para um melhor entendimento, considere a Figura 3.5: à esquerda há um *ranking* de elementos suspeitos, em que dois elementos de programa estão empatados criticamente, e ambos posicionam-se na terceira posição (considerando a estratégia de pior caso). Ao inspecionar os três primeiros elementos, temos que dois deles possuem o mesmo valor de propensão a defeito (e apenas um deles é o defeituoso). Nesse caso, o valor da métrica *actie@3* é 2 (dois). No mesmo exemplo, se aplicarmos *actie@2* o resultado será 1 (um): ao inspecionar os dois primeiros elementos do ranking, somente um deles compartilha o mesmo valor de propensão a defeito com o elemento realmente defeituoso. De forma geral, se dentro do limite descrito por n não houver qualquer elemento empatado criticamente, o *actie* será 0 (zero).

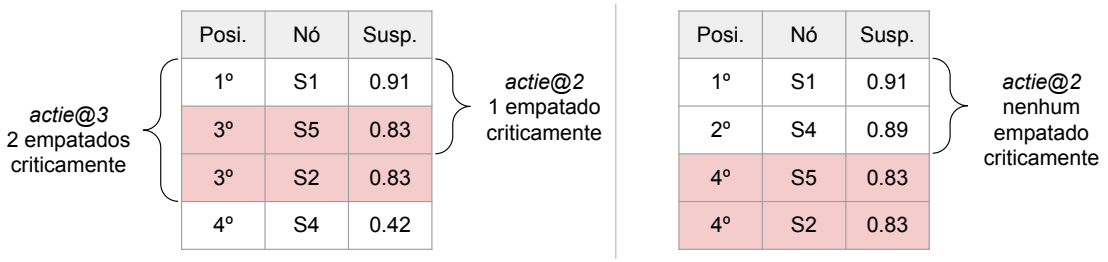

Figura 3.5: Exemplos de *actie* aplicado a rankings, linhas em destaque representam empates críticos.

Assim como a *CScore*, a métrica *actie@n* não expressa eficácia de técnicas de localização de defeitos: é uma métrica para análise complementar, que deve sempre ser utilizada em conjunto com alguma métrica de eficácia. A ideia é investigar como o empate crítico impacta a eficácia da localização de defeitos e, necessariamente, contribuir para o desenvolvimento de métodos em que sua eficácia é *menos dependente* de estratégias de desempate (*tie-break strategies*). Noutras palavras, em geral estratégias de desempate podem agregar imprecisão às medições de eficácia (que é uma ameaça à validade).

Em termos gerais, dois exemplos de aplicação são: a análise conjunta dos valores obtidos pelas métricas $wef@n$ e $actie@n$ pode revelar esforço desperdiçado pelo testador devido a empates críticos; $acc@n$ em conjunto com $actie@n$ pode revelar maior imprecisão da técnica devido à ocorrência de empates críticos, ao inspecionar um número fixo n de elementos de programa. Além de destacar o quanto a técnica analisada poderia ser melhor se dependesse menos de "desempates". O Capítulo 4 inclui uma investigação sobre a aplicação da métrica $actie@n$.

3.5 Considerações finais

Do ponto de vista de localização de defeitos baseada em busca, as duas abordagens introduzidas no presente capítulo exploram regiões próprias no processo de busca. Esses espaços de busca são caracterizados a partir de aspectos tais como: representação da solução, variáveis (operadores), funções objetivo, etc. Dessa forma, ambas abordagens são diferentes entre si e potencialmente geram heurísticas (soluções) distintas daquelas existentes até então.

A primeira abordagem emprega a meta-heurística *Algoritmo Genético* e baseia-se no método apresentado por Wang *et al.* [32]. Novas heurísticas são produzidas a partir de heurísticas originalmente concebidas para o espectro de fluxo de controle. Para explorar operadores atribuídos a heurísticas existentes, a proposta inclui o espectro de fluxo de dados como fonte de informação sobre os defeitos existentes. Assim, novas

heurísticas são produzidas pelo espectro de fluxo de dados e pela colaboração de ambos os espectros (fluxos de controle e de dados).

A segunda abordagem utiliza a meta-heurística *Programação Genética* e baseia-se no método apresentado Yoo *et al.* [40]. Heurísticas são geradas a partir das quatro variáveis de cobertura oriundas da análise de fluxo de controle, em conjunto com oito novas variáveis de cobertura propostas no presente trabalho, as quais são provenientes da análise de fluxo de dados. As equações (heurísticas) são diferentes das propostas até então, pois lidam com uma fonte de informações alternativa às tradicionais (consequentemente, exploram distinto espaço de busca).

Uma nova métrica de avaliação de técnicas de localização de defeitos foi apresentada nesse capítulo. A métrica *empate crítico absoluto – ctie@n* – destaca o quão dependente de estratégias de desempate as técnicas de localização de defeitos são. Representa uma medida absoluta, que é mais próxima do esforço real empregado pelo testador de software em relação a métrica *CScore* apresentada por Xu *et al.* [39], a qual detém um ponto de visto absoluto.

A localização de defeitos automatizada ainda permite muitos avanços. Nesse contexto as abordagens descritas buscam contribuir na depuração em programas com único defeito. Os métodos utilizados para a combinação e criação de heurísticas possuem caráter estocástico, ou seja, pode obter resultados diferentes em cada execução. Por esse motivo, a avaliação utilizada deve ser baseada na análise estatística das execuções.

Enfim, a análise de fluxo de dados pode contribuir diretamente para a localização de defeitos. A aplicação de heurísticas baseadas em *duas* permite encontrar valores de propensão para definições e usos de variáveis. Por exemplo, no método *dua-to-node*, um nó pode receber uma propensão de acordo com as *duas* em que ele participa.

3.6 Prévia de desenvolvimentos futuros

Conforme já mencionado, a Seção 4 complementa as proposições postas no presente capítulo, pela descrição de investigações empíricas das mesmas. Contudo, as abordagens propostas podem ser estendidas, conforme os aspectos elencados a seguir:

- definição de algoritmo para determinar 'como combinar' heurísticas pré-existentes, baseando-se em análise preliminar de um ou mais programas;
- introdução de novas fontes de informação sobre os defeitos; por exemplo, a combinação de elementos de análise estática com os de análise dinâmica da literatura;
- incorporação das oito variáveis de fluxo de dados propostas aos elementos terminais da abordagem implementada por *Algoritmo Genético*;
- integração de ambas as abordagens propostas, tal que a *Abordagem GP* gere novas heurísticas para serem combinadas na *Abordagem GA*.

CAPÍTULO 4

Avaliação das Abordagens

Este capítulo é dedicado à apresentação sobre a análise das abordagens propostas, o que inclui o processo de avaliação e os resultados obtidos.

A Seção 4.1 trata sobre a avaliação da combinação de heurísticas para localização de defeitos baseada em fluxo de dados, e inicia pelas questões de pesquisa que guiam a investigação. A Seção 4.1.1 apresenta os *benchmarks* e *baselines* utilizados, a Seção 4.1.2 introduz os parâmetros para a configuração do experimento. Os resultados são apresentados e discutidos nas Seções 4.1.3 e 4.1.4 e, finalmente, na Seção 4.1.5 são discutidas algumas limitações e ameaças à validade.

A Seção 4.2 lida com a avaliação da criação de heurísticas para a localização de defeitos baseada em fluxo de dados e apresenta, inicialmente, as questões de pesquisa que balizam a análise. A Seção 4.2.1 apresenta os *benchmarks* e *baselines* utilizados, a Seção 4.2.2 é dedicada aos parâmetros de configuração do experimento. Os resultados são apresentados e discutidos na Seção 4.2.3 e, por fim, a Seção 4.2.4 é dedicada a limitações e ameaças à validade.

Na Seção 4.3 são discutidos alguns dos potenciais experimentos futuros, e a Seção 4.4 aborda alguns aspectos adicionais da obtenção de informação de cobertura.

4.1 Avaliação da Abordagem GA – Composição de heurísticas baseada em fluxo de dados

As seguintes questões de pesquisa foram utilizadas para guiar a investigação empírica conduzida a respeito da Abordagem GA – emprego da meta-heurística *Algoritmo Genético* para uma abordagem evolucionária voltada à combinação de heurísticas tradicionais para a localização de defeitos, baseando-se em informação de fluxo de dados, em conjunto com informações de fluxo de controle.

- **Questão de Pesquisa 1 (QP1): A abordagem proposta é competitiva para a localização de defeitos?**

Esta questão direciona a análise da Abordagem GA que utiliza informações de fluxo de dados, em comparação com heurísticas tradicionais e com GA que utiliza informações de fluxo de controle.

- **Questão de Pesquisa 2 (QP2): A combinação evolucionária de informações de fluxo de controle e fluxo de dados contribui para a localização de defeitos?** Esta questão se concentra na investigação do método híbrido, que utiliza ambas fontes de informação sobre o defeito: fluxo de dados e fluxo de controle.

4.1.1 Experimentação

Neste experimento foram utilizados o conjunto de programas *Siemens Suite* [17] conhecidos por serem amplamente utilizados em trabalhos de localização de defeitos [22], e também versões defeituosas do programa *jsoup*[1]. O *Siemens Suite* é composto por 7 programas pequenos em liguagem C, com defeitos artificialmente semeados, enquanto o *jsoup* é escrito em Java e possui defeitos reais. Sendo 112 versões defeituosas dos programas em C, e 36 versões defeituosas do programa em Java. As informações de fluxo de dados para os programas do *Siemens Suite* foram coletadas através de instrumentação manual conforme apresentado na Seção 4.4. Já para o programa *jsoup*, foram utilizados o resultado das execuções da ferramenta JAGUAR disponibilizado pelos autores [2].

A Tabela 4.1 descreve os programas utilizados, mostrando o número de linhas de código (LOC), quantidade de versões defeituosas, e tamanho do conjunto de teste.

Tabela 4.1: Conjunto de programas e versões defeituosas utilizados no experimento.

Programa	LOC	Versões defeituosas	Casos de teste
printtokens	472	6	4030
printtokens2	399	9	4415
replace	512	26	5542
schedule	292	8	2650
schedule2	301	7	2710
tcas	141	35	1608
tot_info	440	21	1051
jsoup	10K	36	468

Baselines

Na avaliação empírica, o termo *baseline* é comumente usado para referenciar abordagens (métodos) que são empregados como referência de comparação ao método

[1] github.com/jhy/jsoup
[2] github.com/saeg/experiments

em análise. Existem diversas heurísticas de localização de defeitos presentes na literatura, dentre as quais as 12 utilizadas nesta experimentação foram escolhidas como *baselines* por: apresentarem bons resultados em trabalhos recentes de localização de defeitos [28, 42]; e estarem entre as mais populares segundo o levantamento de Wong *et al.* [38], sendo elas: *Tarantula* [19], *Ochiai* [1], *OP* [26], *Dstar* [36], *Ample* [9], *Jaccard* [2], *GP*13 [40], *OP*2 [26], *Wong*3 [37], *Zoltar* [18], *Kulczynski*2 [25], e *Barinel* [3].

A Tabela 2.3 apresenta as equações originais dessas heurísticas, e que serão utilizadas na obtenção dos valores de propensão *node* e propensão dua-to-node, conforme definidos na Seção 3.2.1.

4.1.2 Parâmetros do Algoritmo Genético

Na Abordagem GA, o processo de busca envolve determinar conjuntos de pesos que compõem as Equações 3-4, 3-5 e 3-6, a partir de treinamento com versões defeituosas já conhecidas. Esse processo envolve parametrização quanto à busca e ao treino, que deve ser considerada na experimentação.

Ao treinarmos uma solução, ela pode ficar muito especializada no conjunto de versões utilizadas no treino (*overfitting*), e assim não ser generalizável o bastante. Além disso, o processo é estocástico por utilizarmos *Algoritmo Genético* na busca em si, então diferentes resultados podem ser obtidos a cada execução do algoritmo. Essa estocasticidade e o *overfitting* devem ser tratados para aumentar a confiança nos resultados obtidos.

Com intuito de lidar com o *overfitting*, os experimentos foram conduzidos aplicando validação cruzada tripla (*Three-Fold Cross Validation*). Tomando os programas do *Siemens Suite* como exemplo, as 112 versões defeituosas foram divididas aleatoriamente em três subconjuntos disjuntos. Em cada etapa da validação cruzada, um dos subconjuntos é selecionado como *conjunto de validação*, enquanto os demais compõem o *conjunto de treinamento*. Assim, uma solução não é avaliada utilizando as mesmas informações que levaram a compô-la.

Para amenizar o efeito da estocasticidade inerente ao *Algoritmo Genético*, o processo de validação cruzada foi executado 30 vezes. Os resultados estatísticos apresentados referem-se à média dos resultados das 30 execuções dos conjuntos de validação, não computando os resultados obtidos das execuções realizadas essencialmente para treinamento. A Figura 4.1 detalha o processo de experimentação.

Para a implementação do *Algoritmo Genético*, foram aplicadas as configurações básicas do *Framework* DEAP (*Distributed Evolutionary Algorithms in Python*) [10]. Este *framework* contribui no desenvolvimento de algoritmos evolutivos em *python*, permitindo

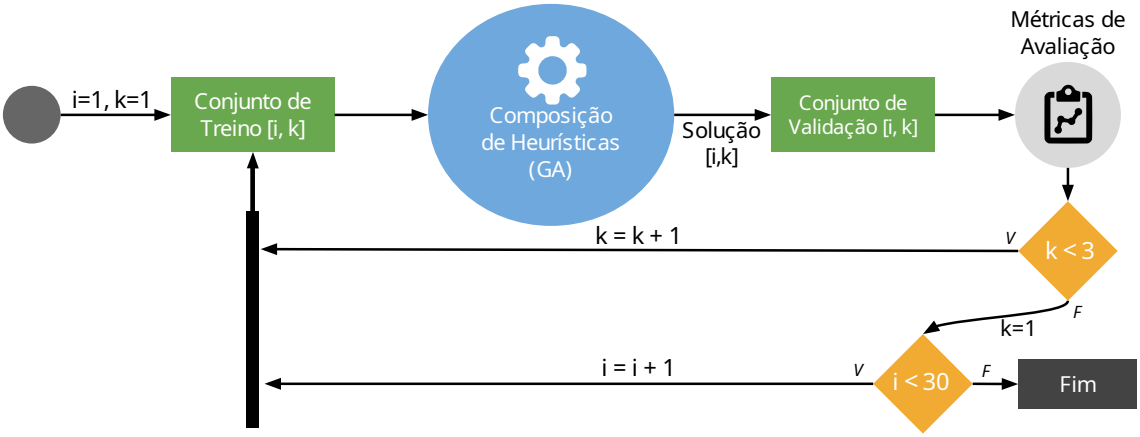

Figura 4.1: Processo de experimentação utilizado na combinação dos valores de propensão a defeito.

a abstração dos operadores básicos. Para a análise estatística foi utilizado o *R Project*[3]

O genótipo dos indivíduos é composto por uma cadeia de 7 bits para cada peso presente na equação, ou seja, 84 bits para as composições simples (HC e $\widehat{HC}$ nas Equações 3-4 e 3-5, respectivamente) e 168 bits para a composição híbrida (HC_{hyb} na Equação 3-6). Foram aplicadas operações de mutação do tipo *Bitflip* e cruzamento de dois pontos de corte. A função *fitness* utilizada foi a mesma empregada por Wang *et al.* em [32]: o *Exam* médio (*AvgExam*) das versões defeituosas contidas no conjunto de validação, representando a média da proporção de código investigado antes de encontrar o primeiro defeito de cada versão defeituosa.

Os demais parâmetros utilizados foram selecionados por meio de experimentos pilotos, em que se buscou valores aos parâmetros com "melhor adaptação" ao problema. Os parâmetros utilizados não foram exaustivamente explorados, entendemos que esta atividade representa, por si só, já uma oportunidade de pesquisa. Ao aplicar, por exemplo, métodos automáticos para esta definição.

Alguns dos parâmetros impactam diretamente no custo de execução da abordagem, nos experimentos pilotos foram levados em consideração um custo aceitável e o desempenho apresentado. Nós executamos os algoritmos para 3 ciclos de validação cruzada (diferentemente dos 30 ciclos da versão final), utilizando por exemplo número de gerações 100, 250, e 500; tamanho da população 25, 50, e 100; Taxa de recombinação 0,6, 0,7,e 0,8; e Taxa de mutação 0,05, 0,01, 0,005

Os parâmetros selecionados são:

- Número de gerações: 250;
- Tamanho da população: 50;
- Taxa de recombinação: 0,6;

[3] r-project.org

- Taxa de mutação: 0,01;
- Seleção: Torneio de tamanho 3.

4.1.3 Resultados

As questões de pesquisa apresentadas anteriormente visam a explorar a capacidade do método proposto em encontrar e diferenciar elementos defeituosos. Quanto à análise dos resultados, alguns pontos merecem destaque:

- Com o objetivo de avaliar a abordagem quanto à sua generalidade, dividimos o conjunto de versões defeituosas apenas entre *Siemens Suite* e jsoup, sem discriminar diferentes programas do *Siemens Suite*.
- Os resultados para as composições HC, $\widehat{HC}$ e HC_{hyb} correspondem ao resultado médio obtido pelas 30 execuções com validação cruzada tripla.
- Em nossa análise, consideramos a estratégia de pior caso (*worst case*) para solução de empates; ou seja, em um *ranking* de suspeitas, se uma mesma propensão a defeito foi atribuída a n comandos e existem m comandos com propensão maior, assumimos que todos os n comandos estarão na posição $n+m$ do *ranking*.

QP1: A abordagem proposta é competitiva para a localização de defeitos?

Nesta avaliação foram utilizadas duas métricas absolutas apresentadas na Seção 2.2.3, a *accuracy* e *wasted effort*, representando, respectivamente: o número de defeitos encontrados investigando n elementos do topo do *ranking*; e o número de elementos não defeituosos investigados, dentre os n elementos, até encontrar o primeiro defeito ao seguir o *ranking*. A discussão a seguir apresenta os resultados com respeito aos cinco primeiros elementos do *ranking* de propensão a defeito, para *accuracy* ($acc@5$) e *wasted effort* ($wef@5$).

Nos gráficos, as heurísticas tradicionais $H(node)$ são representadas por "control-flow", e as heurísticas utilizando a *propensão dua-to-node* $\widehat{H}(node)$ são representadas por "data-flow". Também são apresentados os resultados dos GAs em combinações simples do espectro de fluxo de controle (*GA-cf*) e do espectro de fluxo de dados (*GA-df*), bem como a combinação híbrida (*GA_hyb*) que utiliza ambos os espectros.

A Figura 4.2a mostra os resultados de $acc@5$ para 112 versões defeituosas dos programas do *Siemens Suite*. Neste gráfico, e também nos que virão a seguir, o eixo X traz as heurísticas de localização de defeitos, e o eixo Y é o valor da métrica, $acc@5$ neste caso. Para cada heurística, nós temos duas barras de resultados, a da esquerda mostra os resultados usando fluxo de controle como fonte de informação sobre defeitos, enquanto a da direita mostra os resultados para fluxo de dados. A barra única em GA_hyb mostra o resultado para a combinação de ambas as fontes de informação.

Nos resultados de *acc@5* para o *Siemens Suite*, é possível perceber alguma superioridade em todas as 12 heurísticas individuais de fluxo de dados (de *tarant* até *barine*) em comparação ao fluxo de controle tradicional. Nesse sentido, heurísticas usando fluxo de dados foram capazes de localizar até 36 dos defeitos existentes, contra 33 das heurísticas baseadas em fluxo de controle. Sobre as abordagens evolucionárias baseadas em *Algoritmo Genético*, a composição *GA-cf* foi superior à melhor heurística individual de fluxo de controle (34 contra 33 defeitos localizados); o mesmo se observou para a composição de fluxo de dados *GA-df* (38 contra 36 defeitos localizados). Finalmente a composição híbrida *GA_hyb* superou todas os outros métodos, localizando 39 dos 112 defeitos, inspecionando apenas o *top-5* do *ranking* de propensão a defeito.

A Figura 4.2b apresenta os resultados em relação às 36 versões defeituosas do programa *jsoup*. Assim como na análise do *Siemens Suite*, fluxo de dados superou fluxo de controle em localização de defeitos: heurísticas individuais localizaram até 14 defeitos utilizando fluxo de dados, e 11 utilizando fluxo de controle. A combinação de heurísticas (Abordagem GA) por sua vez só superou as heurísticas individuais quando utilizando fluxo de dados, chegando a encontrar 15 dos defeitos. Controversamente, a abordagem híbrida para a combinação de heurísticas não encontrou mais defeitos do que os outros métodos utilizando fluxo de dados, limitando-se a 14 defeitos localizados.

De acordo com a análise de *acc@5*, heurísticas que se baseiam em informação de fluxo de dados obtiveram resultados competitivos (melhores na maioria dos casos) em relação aos resultados do fluxo de controle. Como a informação de fluxo de dados reflete as relações de definição e uso das variáveis de um programa, a *investigação se torna mais próxima do defeito* se compararmos com fluxo de controle, visto que este utiliza apenas a cobertura de comandos. Essa característica presente em fluxo de dados implica na atribuição de valores de propensão a defeito mais acurados aos comandos do programa.

A Figura 4.3a apresenta os resultados de *wef@5* para o conjunto de programas *Siemens Suite*. Nesta métrica, diferentemente da anterior, resultados com valores menores são melhores. Os métodos que utilizam informação de fluxo de controle obtiveram resultados levemente melhores em relação aos que utilizam fluxo de dados. Por exemplo a heurística *OP2* que, até encontrar o defeito ou alcançar o limite ($n = 5$), precisou inspecionar 440 elementos usando fluxo de controle e 455 usando fluxo de dados. A composição *GA-cf* apresenta uma pequena melhoria em comparação às heurísticas individuais, reduzindo apenas um elemento inspecionado (439 contra 440 elementos). A composição *GA_hyb* é a que demonstra melhor resultado, superando todos os outros métodos analisados, demandando a inspeção de 416 elementos em todas as 112 versões defeituosas, até localizar os defeitos ou atingir o limite ($n = 5$).

A Figura 4.3b traz os resultados de *wef@5* para o programa *jsoup*. Diferentemente do que acontece para o *Siemens Suite*, para o *jsoup* os métodos baseados em infor-

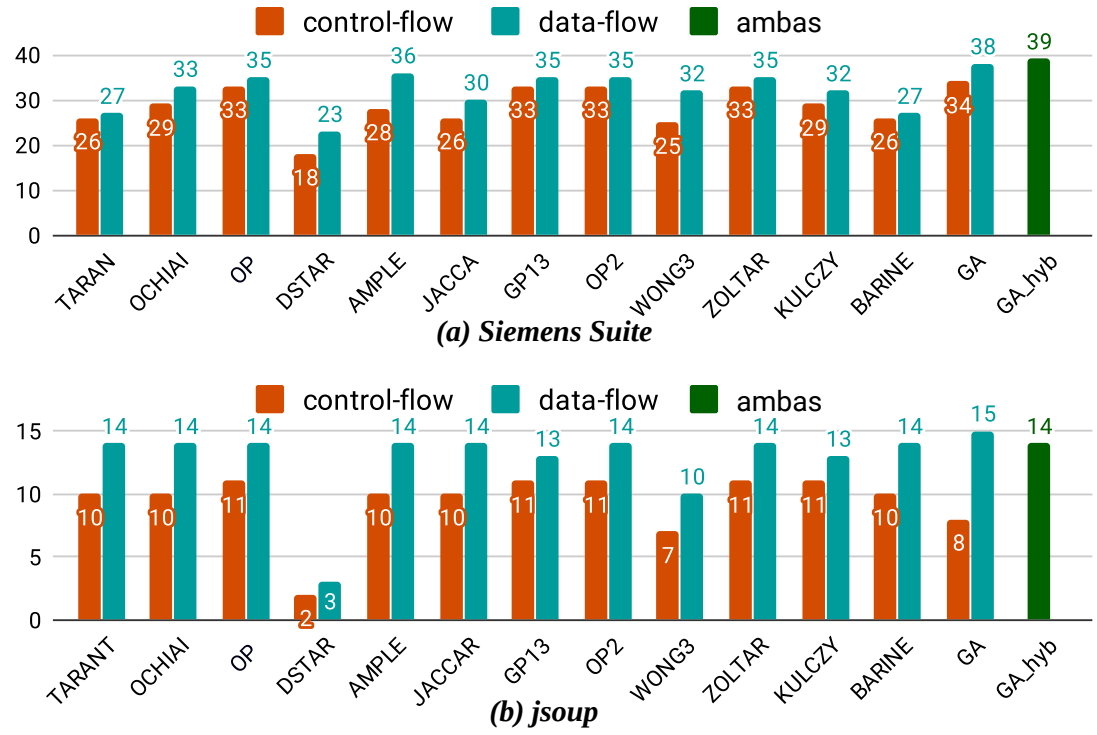

Figura 4.2: Resultados para *Accuracy* (*acc@5*).

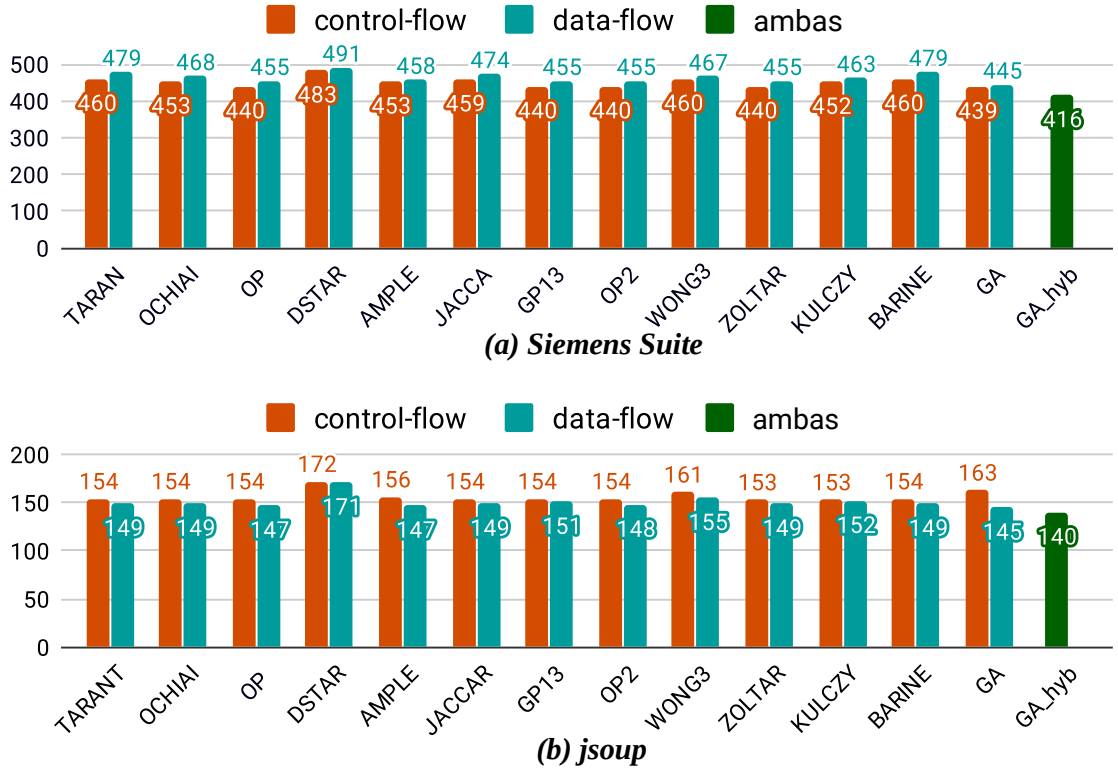

Figura 4.3: Resultados para *Wasted effort* (*wef@5*).

mação de fluxo de dados performaram melhor. Por exemplo a heurística *Ample* demandou

a inspeção de 147 elementos quando utilizando fluxo de dados, e 156 elementos quando utilizando fluxo de controle. A composição *GA-cf* teve resultados semelhantes às heurísticas individuais, já a composição *GA-df* alcançou leve melhoria. A composição híbrida (*GA_hyb*) trouxe o melhor resultado dentre todos métodos analisados (140 elementos investigados), dessa forma foi consistente em relação ao avanço apresentado para o *Siemens Suite*.

Os resultados *wef* são menores quando o número de defeitos localizados (*acc*) é maior. Este comportamento pode ser o motivo para os métodos baseados em fluxo de dados apresentarem melhores valores de *wef* para o programa *jsoup*. Porém tal comportamento demonstra ser distinto para o caso do *Siemens Suite*. Para auxiliar a interpretação desses resultados, um aspecto importante sobre a *propensão dua-to-node* deve ser considerado: essa abordagem tem a característica intrínseca de atribuir o mesmo valor de suspeita para todos os comandos que participam da mesma associação de fluxo de dados (*dua*).

QP2: A combinação evolucionária de informações de fluxo de controle e fluxo de dados contribui para a localização de defeitos?

De acordo com os resultados da métrica *accuracy* apresentados, é notável a habilidade para localizar defeitos presente em métodos baseados em fluxo de dados, a *propensão dua-to-node* foi capaz de alcançar resultados que não seriam alcançados usando fluxo de controle. O GA também produziu avanços mais significantes quando aplicado à fluxo de dados. Embora a abordagem híbrida demonstre uma pequena vantagem no conjunto de programas *Siemens Suite*, não é possível notar o mesmo acontecendo para o programa *jsoup*, onde ela apenas mantém resultados "equiparáveis" aos melhores métodos analisados. Por outro lado, a composição *GA_hyb* expressa uma contribuição relevante quanto à Métrica *wasted effort*. Assim sendo, *GA_hyb* apresenta-se competitiva em *acc@5*, com resultados próximos aos das heurísticas utilizando fluxo de dados, mas demandando uma quantidade de elementos inspecionados inferior.

Como dito anteriormente, a *propensão dua-to-node* possui potencial para atribuir o mesmo valor de suspeita para vários elementos de programa: a todos os nós componentes de uma *dua*, é atribuída a mesma propensão de defeito dessa *dua*, e uma estratégia é necessária para computar a propensão de um nó, quando este é componente de várias *duas* distintas. Assim, uma investigação é pertinente para: (i) analisar o impacto dessa "tendência" da *propensão dua-to-node* (vários elementos de programa com o mesmo valor de suspeita); e (ii) examinar se a composição híbrida atenua essa inclinação.

Para conduzir essa investigação, utilizamos a métrica proposta na Subseção 3.4: empate crítico absoluto (*actie@n*). A Figura 4.4 mostra um resumo dos resultados de *acc*, *wef*, e *actie*. As melhores heurísticas em *accuracy* foram selecionadas para representar

todas as heurísticas analisadas, para o *Siemens Suite* foram a *Zoltar* com fluxo de controle (*Zoltar-cf*) e *Ample* com fluxo de dados (*Ample-df*). Já para o *jsoup*, foram escolhidas Zoltar com fluxo de controle (*Zoltar-cf*) , e também com fluxo de dados (*Zoltar-df*).

Considerando o *Siemens Suite* na Figura 4.4a, os métodos baseados em fluxo de controle – *Zoltar-cf* e *GA-cf* – apresentam resultados similares para *actie* (96 e 97, respectivamente). Essa similaridade também ocorre entre os métodos baseados exclusivamente em fluxo de dados (*Ample-df* e *GA-df*, com valores 136 e 134, respectivamente), contudo com números superiores aos de fluxo de controle. A composição híbrida (*GA_hyb*) produz o melhor valor para *actie* (igual a 70) dentre todas as abordagens. No *jsoup* (Figura 4.4b), o resultado de *actie* para métodos baseados exclusivamente em fluxo de dados é maior do que os baseados em fluxo de controle, o que está em sintonia com o obtido no *Siemens Suite*. Em adição, a composição híbrida trouxe, novamente, resultados competitivos com respeito à métrica *actie*.

As constatações do parágrafo anterior confirmam que: (i) a *propensão dua-to-node* tem maior empate crítico absoluto do que à *propensão node*; e (ii) a composição híbrida, que envolve informações dos fluxos de controle e de dados, atenua essa tendência da *propensão dua-to-node*.

Apesar de serem melhor avaliados quanto à *accuracy*, métodos que utilizam unicamente a *propensão dua-to-node* apresentam mais empates críticos do que métodos baseados exclusivamente em fluxo de controle. *GA_hyb* consegue reduzir a dependência por estratégias de desempates, e ainda manter uma eficácia satisfatória proveniente da informação de fluxo de dados, demonstrada pelos seus bons resultados para *accuracy* e *wasted effort* em ambos *benchmarks*. Dessa forma, o uso de informação de fluxo de dados agrega valor à localização de defeitos, mas resulta em maior dependência de estratégias de desempate com respeito à propensão de defeito, o que pode ser reduzido pelo emprego de abordagens híbridas.

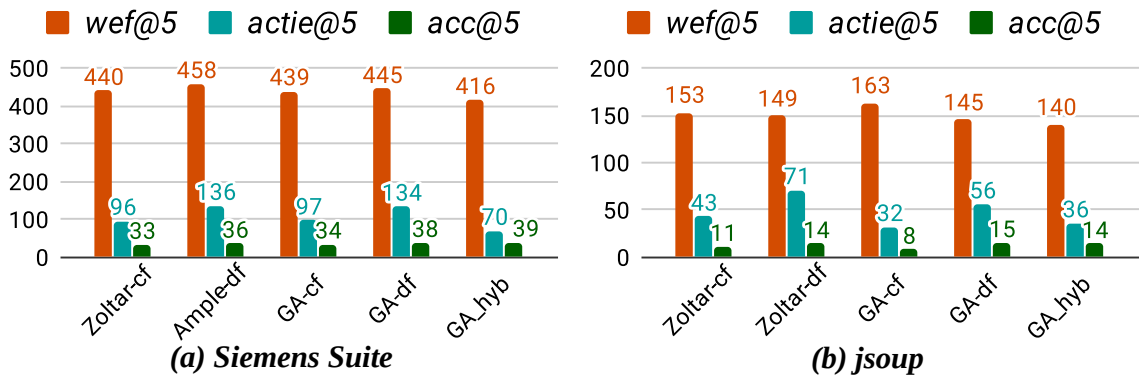

Figura 4.4: Resultados para empate crítico absoluto (*actie@5*) em relação aos resultados de *Wasted effort* (*wef@5*) e *Accuracy* (*acc@5*).

4.1.4 Análise estatística

Para lidar com a estocasticidade presente nas abordagens evolutivas e aumentar a confiança dos resultados obtidos, foram aplicados dois testes estatísticos, a respeito dos resultados de 30 execuções. A comparação entre pares *Wilcoxon* e testes *Vargha & Delaney* $\hat{A}_{12}$, conforme recomendado por Arcuri e Briand [6]. O primeiro teste é usado para revelar se há diferença com significância estatística entre os resultados produzidos pelos diferentes métodos. O último mostra o quão superior são os resultados de um método em comparação a outros.

A Tabela 4.2 mostra os resultados para os testes estatísticos comparando os valores de *acc*, *wef* e *actie* obtidos por todos os métodos que foram utilizados, *GA_hyb*, *GA-cf* (GA baseado em fluxo de controle), *GA-df* (GA baseado em fluxo de dados), *H-cf* (heurística aplicada em fluxo de controle), e *H-df* (heurística aplicada em fluxo de dados). Para *H-cf* e *H-df*, foram selecionadas as heurísticas com melhores resultados para cada métrica, exceto para *actie* que não expressa qualidade independentemente.

Tabela 4.2: Resultados para testes Vargha & Delaney $\hat{A}_{12}$ considerando os métodos de localização de defeitos e todas as métricas utilizadas

		Siemens Suite			**jsoup**		
		GA_hyb	GA-cf	GA-df	GA_hyb	GA-cf	GA-df
GA-cf	acc@5	0.00			0.00		
	wef@5	1.00			1.00		
	actie@5	1.00			0.14		
GA-df	acc@5	0.15	1.00		0.82	1.00	
	wef@5	1.00	0.99		0.88	0.00	
	actie@5	1.00	1.00		1.00	1.00	
H-cf	acc@5	0.00	0.13	0.00	0.00	0.98	0.00
	wef@5	1.00	0.80	0.00	1.00	0.00	1.00
	actie@5	1.00	0.18	0.00	1.00	1.00	0.00
H-df	acc@5	0.00	1.00	0.00	**0.53**	1.00	0.03
	wef@5	1.00	1.00	1.00	1.00	0.00	0.83
	actie@5	1.00	1.00	1.00	1.00	1.00	1.00

No programa *jsoup*, *H-cf* representa *Zoltar* para as três métricas, *H-df* representa *Zoltar* para *acc* e *actie*, e *Ample* para *wef*. Já no conjunto de programas *Siemens Suite*, *H-cf* representa *Zoltar* para as três métricas, mas *H-df* representa *Ample* para *acc* e *Zoltar* para *wef* e *actie*.

Não foi executada comparação estatística entre *H-cf* e *H-df*, pois ambos são métodos determinísticos. Os valores descritos na Tabela 4.2 se referem aos testes Varha & Delaney $\hat{A}_{12}$, onde 1 é o método na linha, e 2 é o método na coluna. O valor em **negrito** indica que não houve diferença estatisticamente significante, considerando o nível de confiança de 99%.

No caso do teste *Wilcoxon*, seu resultado é o *p-value*, o qual representa a chance de duas amostras serem estatisticamente iguais. Isso significa que mesmo quando os

gráficos mostram resultados "próximos", existe diferença estatisticamente significante entre os métodos. Nesta análise, houve diferença estatística em todas as comparações, exceto uma, que está apresentada em **negrito** na Tabela 4.2.

Quanto aos testes $\hat{A}_{12}$, seu resultado mostra com que frequência os valores do método 1 (linha) são superiores aos do método 2 (coluna). Quanto mais perto de 1.0, mais altas são as chances do valor do método 1 superar o método 2. Da mesma forma que resultados mais próximos de 0.0 indicam a maior chance do método 1 ser inferior ao método 2. Por exemplo, considere a comparação *GA-cf* em relação a *GA_hyb*, na métrica *acc@5*, o teste estatístico resultou *0.0* (*i.e. GA-cf* é sempre inferior à *GA_hyb*). Para a métrica *acc@5* valores maiores são melhores, então podemos afirmar com maior confiança que o *GA_hyb* é melhor que *GA-cf* nesta métrica. Para o caso da métrica *wasted effort* valores próximos de *0.0* indicarão superioridade, visto que nesta métrica valores menores são melhores.

Quanto à *accuracy*, no *Siemens Suite GA_hyb* se mostrou superior a todos os outros métodos, já no *jsoup* ele foi inferior à *GA-df* e não apresentou diferença estatística quando comparado à *H-df*.

Quanto à *wasted effort*, os resultados dos testes $\hat{A}_{12}$ mostram que *GA_hyb* supera todos os outros métodos, já que valores menores são melhores para esta métrica. Para empate crítico absoluto, a abordagem híbrida só ficou atrás de *GA-cf*, e somente na análise do programa *jsoup*

Assim sendo, considerando os resultados apresentados nesta seção, é possível responder as duas questões de pesquisa.

QP1: A abordagem proposta é competitiva para a localização de defeitos? Com respeito à *accuracy*, a *propensão dua-to-node* se demonstrou melhor do que as heurísticas baseadas em fluxo de controle no *Siemens Suite* e também no *jsoup*, mas ao analisarmos *wasted effort*, o mesmo não acontece para o conjunto de programas *Siemens Suite*. Além disso o número de empates críticos é consideravelmente maior quando comparado com heurísticas tradicionais, em ambos os *benchmarks*. Então a resposta para QP1 é: Sim, a abordagem proposta é competitiva para localização de defeitos, apresentando resultados melhores do que métodos tradicionais, quando analisado o número de defeitos localizados investigando um número limitado (pequeno) de elementos.

QP2: A combinação evolucionária de informações de fluxo de controle e fluxo de dados contribui para a localização de defeitos? A abordagem *GA_hyb* demonstrou localizar tantos defeitos quanto às heurísticas *propensão dua-to-node*, além de reduzir o esforço perdido e o número de empates críticos. Então a resposta para a QP2 é: Sim, a abordagem híbrida apresenta contribuições por manter resultados de localização de defeitos compatíveis com os melhores métodos analisados, e ainda diminuir o número de elementos inspecionados e o número de empates críticos, reduzindo sua dependência

de estratégias de desempate.

4.1.5 Limitações e ameaças à validade

Alguns aspectos da avaliação por experimentos podem oferecer potenciais ameaças à validade dos resultados obtidos, conforme mencionados a seguir.

Ameaças de conclusão – os resultados das abordagens evolutivas sofrem de estocasticidade, ou seja, podem ser diferentes em cada execução. Para tratar essa ameaça, foram conduzidas 30 execuções de cada configuração do GA, e aplicados testes estatísticos para contribuir com a confiança dos resultados.

Ameaças internas – os três métodos baseados em GA foram executados usando os mesmos parâmetros, como taxa de mutação, taxa de cruzamento e número de gerações. As informações de fluxo de controle e fluxo de dados do conjunto programas *Siemens Suite*, que foram obtidas através de instrumentação manual, foram extensivamente revisadas pelo autor e pelo orientador da pesquisa, e suas saídas foram comparadas com as saídas originais. Os programas originais e instrumentados estão disponíveis publicamente[4]. O espectro de cobertura do programa *jsoup* foi obtido através da ferramenta *Jaguar* e também estão disponíveis online[5].

Ameaças de construto – a nova métrica de avaliação (*empate crítico absoluto* – $actie@n$), que é direcionada à medição da dependência de estratégias de desempate, foi aplicada em conjunto com métricas da literatura voltadas à eficácia. Essa aplicação conjunta fortalece a confiança nos números obtidos pela $actie@n$, pois demonstram consistência com os resultados de eficácia.

Ameaças externas – para resultados mais generalizáveis, é necessário uma investigação em grande escala, incluindo programas de diferentes linguagens de programação. Apesar de não ser uma investigação com grande número de versões defeituosas, os programas utilizados nesse experimento foram escolhidos buscando abranger diferentes contextos, incluindo programas de diferentes tamanhos, linguagens de programação, com defeitos reais e semeados.

4.2 Avaliação da Abordagem GP – Geração de heurísticas baseadas em fluxo de dados

A seguir será discutida a avaliação preliminar da abordagem GP: geração de heurísticas de localização de defeitos pelo algoritmo *Programação Genética*, a partir

[4] github.com/I4Soft/df-evolutionary-fl
[5] github.com/saeg/experiments

de variáveis de fluxo de dados. Utilizando novas variáveis de cobertura, obtidas das informações de fluxo de dados, é possível gerar novas equações nunca antes utilizadas para a localização de defeitos. Esta abordagem ainda não foi publicada por qualquer veículo com revisão por pares, mas já é possível discutir seus resultados promissores. Para esta investigação, buscamos responder a seguinte questão de pesquisa:

- **A informação de fluxo de dados contribui para a eficácia e eficiência de abordagens de localização de defeitos baseadas em Programação Genética?** Essa questão busca explorar o método proposto não apenas quanto ao incremento de qualidade das soluções (eficácia), mas também quanto ao custo (eficiência) para manter a qualidade competitiva.

4.2.1 Experimentação

Assim como na avaliação da abordagem anterior, foi utilizado o conjunto de programas *Siemens Suite* que é formado por 7 programas em C, pequenos e com defeitos semeados artificialmente. Foram utilizadas 112 versões defeituosas como apresentado na Tabela 4.3. As informações de cobertura de fluxo de controle e fluxo de dados foram obtidas através de instrumentação manual, apresentada na Seção 4.4. Como *baselines* foram utilizadas as heurísticas tradicionais *Ochiai* e *Tarantula*, além de uma abordagem evolucionária GP baseada em variáveis de cobertura de fluxo de controle.

Tabela 4.3: Conjunto de programas e versões defeituosas utilizados no experimento.

Programa	LOC	Versões defeituosas	Casos de teste
printtokens	472	6	4030
printtokens2	399	9	4415
replace	512	26	5542
schedule	292	8	2650
schedule2	301	7	2710
tcas	141	35	1608
tot_info	440	21	1051

4.2.2 Parâmetros da Programação Genética

Para esta experimentação, os métodos baseados em GP foram executados com a seguinte configuração (parâmetros selecionados por experimentos pilotos):

- Taxa de mutação de único ponto: 0,07;
- Taxa de mutação de subárvore: 0,03;
- Taxa de cruzamento: 0,80;
- Taxa de reprodução: 0,10;

- Funções aritméticas: adição, subtração, multiplicação e divisão “segura”;
- Terminais: variáveis de cobertura e constantes $[-1, 0, 1]$;
- Tamanho da população: 100;
- Altura máxima da arvore: 4.

As funções aritméticas e as constantes terminais selecionadas são as mesmas utilizadas por *Yoo* [40], a altura máxima da arvore foi selecionada levando em consideração o tamanho das heurísticas tradicionais, e o número de folhas em relação ao número de variáveis possivelmente utilizadas. As taxas de mutação, cruzamento e reprodução foram escolhidas de forma que a soma não ultrapassasse 1, as variações executadas foram em torno das mutações de único ponto e subárvore. Visto que a mutação de subárvore representa grandes modificações nos indivíduos quando a taxa é muito alta, as soluções não convergem para um mesmo fitness. Por outro lado, uma nova subárvore pode também ter um impacto positivo nas soluções. Portanto foram escolhidos as taxas que melhor permitiram a diversificação das soluções encontradas, sem comprometer a intensificação.

A função *fitness* utilizada é *Exam* médio, ou seja, a média da proporção de código investigado até encontrar o primeiro defeito de cada versão defeituosa do conjunto de treino. Novamente foi utilizada a estratégia de validação cruzada de tamanho 3, apresentada na Figura 4.5. Para lidar com a estocasticidade presente na abordagem, os resultados referem-se à média dos números obtidos em 30 execuções. Para implementar as GP foi utilizado o framework DEAP [6], e o R Project[7] para a análise estatística.

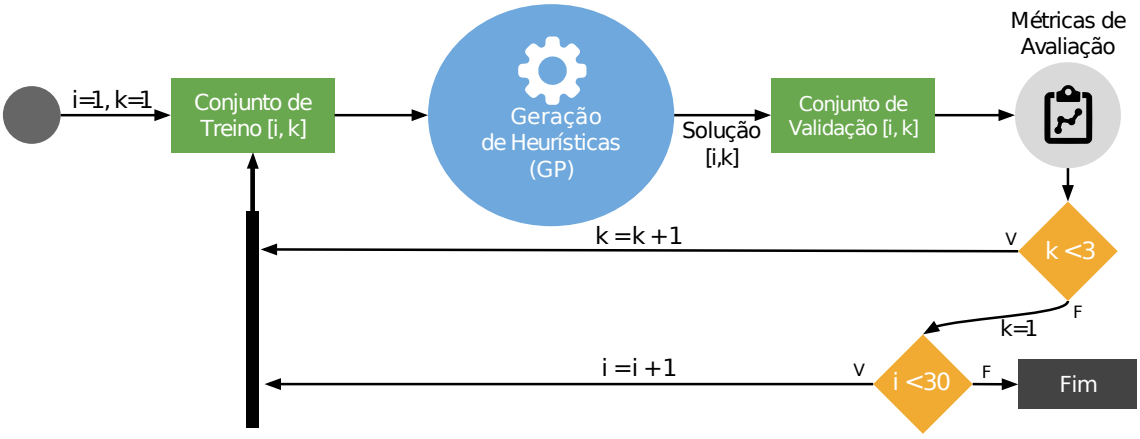

Figura 4.5: Processo de experimentação utilizado na geração de novas equações para localização de defeitos.

[6] http://deap.readthedocs.io
[7] https://www.r-project.org

4.2.3 Resultados

Foram usadas duas métricas de valores absolutos para a análise da eficácia: *accuracy (acc@n)*, que mensura o número de defeitos encontrados considerando o top *n* no *ranking* de suspeita (valores maiores são melhores); e *wasted effort (wef@n)*, que mede o número de elementos investigados antes da localização de um defeito, limitado a *n* elementos para cada versão defeituosa (valores menores são melhores).

A Figura 4.6 apresenta o *acc* e *wef* para $n = 10$. O eixo X representa as gerações da GP. Os valores de *acc* e *wef* (eixo Y) foram obtidos para o melhor indivíduo de cada geração de acordo com a função *fitness*. Os gráficos apresentam: *GP-cf* utilizando quatro variáveis de cobertura de fluxo de controle; *GP-df* com oito variáveis de cobertura de fluxo de dados; e *GP-hyb* híbrido com 12 variáveis dos fluxos de controle e de dados. As linhas constantes mostram o valor das heurísticas *Tarantula* e *Ochiai*.

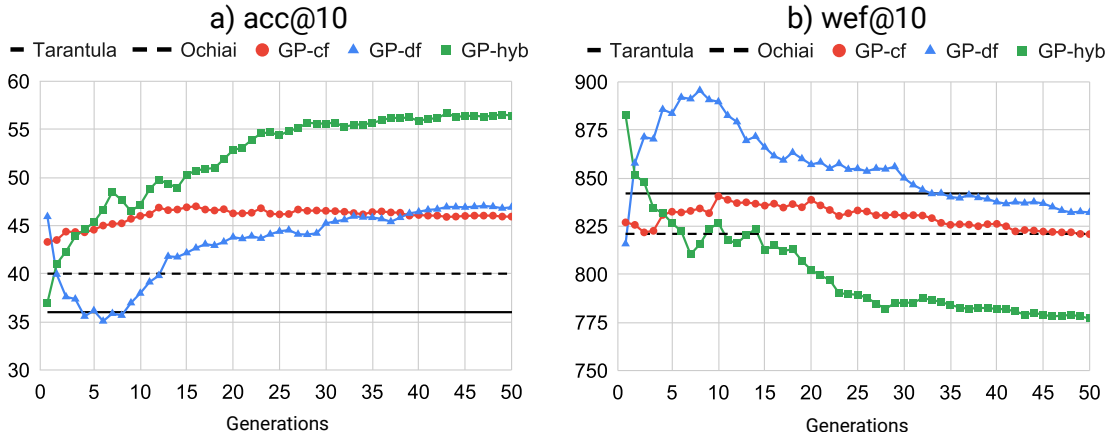

Figura 4.6: Valores de *acc@n* e *wef@n* com respeito aos dez ($n = 10$) elementos mais suspeitos em cada versão defeituosa.

QP: A informação de fluxo de dados contribui para a eficácia e eficiência de abordagens de localização de defeitos baseadas em programação genética?

Para melhor responder esta questão, os resultados foram divididos entre eficácia e eficiência, cada uma abordando uma dimensão própria dos resultados.

Eficácia

Na Figura 4.6-a, que representa o *acc*, é possível observar que todas as abordagens baseadas em GP superaram as heurísticas tradicionais no curso das gerações. Enquanto a *GP-cf* e a *GP-df* demonstraram resultados "próximos", a *GP-hyb* apresentou visíveis melhorias em comparação com todos os demais métodos. Nos resultados de *wef* na Figura 4.6-b, há um comportamento similar, exceto pelo fato de a *Ochiai* ter apresentado resultados superiores aos *GP-df*, e ter alcançado quase os mesmos resultados que a

GP-cf ao fim de 50 gerações. Novamente, a *GP-hyb* teve a melhor performance dentre os métodos analisados.

É possível analisar o *fator de precisão* dos métodos utilizando o $acc@10/wef@10$ representando uma média de elementos inspecionados para cada defeito encontrado. Assim, para *GP-hyb* foi observada uma relação de 13,87 (777/56) elementos inspecionados para cada defeito localizado, enquanto *GP-cf* obteve 17,82 (820/46) e *GP-df* 17,70 (832/47), já a *Ochiai* obteve uma média de 20,52 (821/40), e *Tarantula* 23,39 (842/36).

Tais resultados indicam que a *GP-hyb*, ao fim de 50 gerações, consegue encontrar equações que localizam mais defeitos e demandam menos elementos inspecionados, em relação aos demais métodos do experimento. Em adição, o comportamento de crescimento (*acc*) e de queda (*wef*) ao longo das gerações sugere que as abordagens baseadas em GP que utilizam fluxo de dados aparentam ser mais adequadas à evolução, já que demonstram melhoria relativa dos resultados com respeito às demais.

Eficiência

Antes de focar em números que medem a eficiência, é pertinente destacar a degradação da eficácia (*acc* e *wef*) nas primeiras gerações da série *GP-df*– comportamento observado na linha azul das Figuras 4.6-a e 4.6-b. Nós atribuímos este comportamento à diferença entre: o objeto das métricas de avaliação (*acc* e *wef*); e o objeto da métrica de treinamento (a função *fitness*, que guia a aprendizagem). O primeiro refere-se à uma medida absoluta, que evoca o número de elementos investigados para a localização de defeitos; o segundo também lida com tal número, contudo normalizado pelo tamanho do software (medida relativa). Dessa forma, o treinamento é influenciado pelo tamanho do software, o que não ocorre na etapa de validação. A Figura 4.7 apresenta o valor médio da fitness no passar das gerações. A função *fitness* expressa a proporção média do código investigado (valores menores são melhores). Assim, percebemos que não há detrimento da fitness pra *GP-df* através das gerações como vimos na Figura 4.6-a, o que corrobora com a nossa conjectura. É possível que a redução da proporção de código investigado média, prejudique o resultado nas métricas absolutas.

Para medir o custo da Abordagem GP, destacamos que, em geral, o custo essencial das abordagens evolucionárias situa-se na fase de treinamento. Nesse sentido, a redução do número de gerações pode abreviar o esforço de treinamento e, portanto, reduzir o custo da abordagem.

Na Figura 4.6-a, *GP-cf* apresenta resultados melhores que as heurísticas individuais (*Tarantula* e *Ochiai*) logo nas primeiras gerações, atingindo cedo (Geração 16) sua melhor *accuracy* ($acc@n = 47$), e então decrescendo até sua estabilização ($acc@n = 44$). Este comportamento pode indicar a ocorrência de *overfitting*. Para o *wasted effort* na Fi-

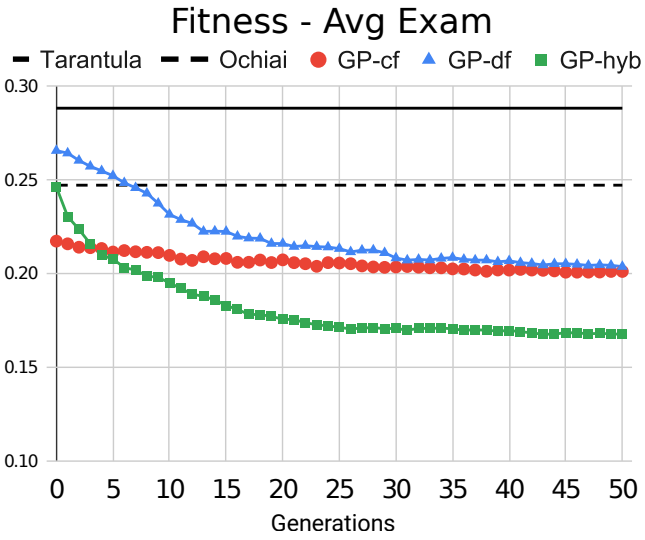

Figura 4.7: Valor médio da fitness (Avg Exam) através das gerações, valores menores são melhores.

gura 4.6-b, nota-se que a *GP-cf* alcança a *Ochiai* na Geração 3, em seguida não estabiliza nem apresenta melhoria até a Geração 50.

No caso da *GP-df*, após a degradação de *accuracy* e *wasted effort* nas primeiras gerações, observa-se melhoria com relação a ambas métricas nas gerações seguintes. Logo na Geração 12, há superioridade de *accuracy* em relação a *Ochiai* e *Tarantula*. No caso do *wasted effort*, *GP-df* não supera *Ochiai* após 50 gerações. Este comportamento pode, também, ser atribuído à diferença entre os objetos das métricas de treinamento e de avaliação.

Com relação à abordagem *GP-hyb*, o custo para superar (ou alcançar) a *accuracy* das demais é: zero gerações para *Tarantula*; duas gerações para *Ochiai* e *GP-df*; e sete gerações para *GP-cf*. Sobre o *wasted effort*, a abordagem *GP-hyb* é melhor que *Tarantula* e *Ochiai* após sete gerações; importante ressaltar que *GP-hyb* supera todas as *baselines* após 15 gerações, somente.

Discussão

Foi verificada diferença estatística significante entre *GP-hyb* e todos os demais métodos no teste *Wilcoxon signed-rank* com nível de confiança de 99%; o teste *effect size* demonstra que a *GP-hyb* foi superior com grande vantagem.

Ao atribuir valores de suspeita aos elementos do código, elementos distintos podem receber o mesmo valor de suspeita. Este evento é chamado de empate e é um obstáculo comum na localização de defeitos. Quando são utilizadas novas variáveis — obtidas de outras fontes de informação sobre os defeitos – para o cálculo da suspeita, a frequência da ocorrência de empates pode ser reduzida. Tal cenário impacta na efetividade

do método, pois resulta em menor dependência relativa de estratégias de desempate. Finalmente, podemos responder à questão de pesquisa.

QP: A informação de fluxo de dados contribui para a eficácia e eficiência de abordagens de localização de defeitos baseadas em programação genética?
Em relação à *accuracy* e *wasted effort*, as informações de fluxo de dados individualmente apresentaram resultados similares em comparação com fluxo de controle ao final de 50 gerações, mas sem melhoria na eficiência. No entanto, a abordagem híbrida teve melhor desempenho em relação a todos os outros métodos em 50 gerações, além de resultar em avanço considerável de qualidade em apenas 15 gerações.
Desta forma, a resposta para a QP é: *Sim, informações de fluxo de dados, quando utilizadas em conjunto com fluxo de controle, contribuem tanto para a eficácia quanto para a eficiência de abordagens baseadas em Programação Genética para localização de defeitos, detectando mais defeitos e demandando poucas gerações para tal, em relação aos baselines.*

Um ponto importante encontrado nessa investigação remete à utilização de métricas de eficácia em termos absolutos, como *Accuracy* e *Wasted Effort*, e métricas de eficácia em termos relativos como o *Exam*. Percebeu-se que em alguns cenários elas se mostram conflitantes, apesar de ainda expressarem eficácia.

Conduzimos investigações preliminares utilizando *Acc* como função fitness, mas percebemos que métricas absolutas, do jeito que são apresentadas hoje, não contribuem na evolução dos métodos evolucionários, isso porque elas tendem à não variar seus valores para indivíduos pouco diferentes. Dificultando assim a diferenciação dos indivíduos e a seleção dos mais aptos.

4.2.4 Limitações e ameaças à validade

Alguns aspectos desta experimentação podem ser interpretados como potenciais ameaças à validade. Foram conduzidas 30 execuções para cada configuração da GP, com a intenção de mitigar o problema da estocasticidade, além disso foram aplicados testes estatísticos para aumentar a confiança dos resultados obtidos.

Todas as abordagens que usam GP foram executadas sob os mesmos parâmetros, inclusive de altura da árvore da representação das possíveis soluções. Porém algumas possuem maior número de variáveis (elementos terminais para a árvore) do que outras, gerando assim um espaço de busca bastante diferente. Tal característica pode ocasionar espaços de busca mais promissores para algumas das abordagens GP.

O conjunto de programa utilizado na avaliação da abordagem, apesar de bastante utilizado na literatura, não é bastante representativo. Portanto, a extensão da presente investigação requer, necessariamente, a expansão para outros *benchmarks*.

4.3 Prévia de experimentos futuros

Durante a execução das investigações apresentadas, consideramos algumas oportunidades ainda não exploradas, algumas dessas são listadas abaixo.

O **emprego de novas funções *fitness*** é relevante devido ao uso de métricas com naturezas distintas para as fases de treinamento e avaliação (métricas relativas e absolutas, respectivamente). Dessa forma, é pertinente investigar o uso de diferentes métricas como função *fitness*, inclusive combinando-as. Um primeiro passo poderia ser o *fator de precisão* apresentado na Seção 4.2.3

A investigação do impacto no **custo de execução** foi aplicada apenas à abordagem GP. Também é pertinente estender a análise de custo à abordagem GA. De maneira similar, a investigação quanto à **dependência de estratégias de desempate** foi aplicada apenas à abordagem GA, sendo ainda necessária a investigação no contexto da GP.

A **agregação de novos benchmarks** se faz necessária para alcançar programas grandes com defeitos reais, tal como o *Benchmark* Defects4J [20]; este *benchmark* possui programas em Java, com defeitos reais, que são amplamente utilizados no contexto de localização de defeitos.

4.4 Aspectos de Implementação

Para obter informações de fluxo de dados em programas C, existem ferramentas como Poke-Tool [8] e ATAC [23], porém são ferramentas descontinuadas e que não oferecem dados de cobertura sobre essas informações. Assim, os programas defeituosos em C utilizados foram instrumentados manualmente. A abordagem utilizada desconsidera linhas que não possuem comandos, além de identificar individualmente comandos em uma mesma linha.

Nesse contexto, apenas os nós e *duas* exercitados pelos casos de teste possuem a informação de interesse (cobertura), assim, os "instrumentos" utilizados foram construídos com intuito de gerar arquivos de *log* das execuções dos casos de teste, sem alterar o resultado do código original. Um caso de teste cobre um nó ou *dua* se isto é descrito no *log* de sua execução. Os "instrumentos" e as informações que são capturadas por eles são:

- **_i_node**. Registra a execução de um nó. Contribui com informações referentes aos comandos alcançados na análise de cobertura de nós;
- **_i_def**. Registra o nó de definição de uma variável, ao ser combinado com um uso desta mesma variável, gera uma *dua*;
- **_i_cuse**. Registra o nó de uso computacional de uma variável;
- **_i_puse**. Registra a variável usada em uma bifurcação do grafo de fluxo de controle;

- **_i_loc**. Registra o local de uso de um _i_puse. Todo _i_loc está relacionado com um ou mais _i_puse.

Através dos *logs* gerados pela execução dos códigos instrumentados é possível obter todos os comandos e *duas* exercitados pelos casos de teste, e assim medir propensão a defeitos de ambos para o mesmo programa. O Apêndice A traz um exemplo de código utilizando a instrumentação apresentada.

CAPÍTULO 5

Considerações Finais

Este trabalho apresenta o estudo conduzido sobre o uso de informação de fluxo de dados para computar valores de propensão à defeitos de elementos de código, no contexto de algoritmos evolucionários para a localização de defeitos em software. Foram introduzidas duas abordagens principais que se baseiam em meta-heurísticas, *Algoritmo Genético* e *Programação Genética*, para a combinação de heurísticas existentes e a geração de novas heurísticas, respectivamente.

Fluxo de dados utiliza da informação de relações entre local de definição e local de uso de variáveis; tal relação é também denominada associação def-uso. Por conter informação "mais próxima" do defeito, era esperado que heurísticas alicerçadas em informação de fluxo de dados apresentassem bons resultados para localização de defeitos. Mas a utilização da análise de fluxo de dados nesse contexto não é tão natural quanto fluxo de controle: cada elemento de fluxo de dados pode incluir até três comandos (nós). Assim, foi apresentada uma estratégia para traduzir valores de suspeita atribuídos à associações definição-uso, para os comandos que participam destas associações, a *propensão dua-to-node*.

Com a *propensão dua-to-node* foi possível aplicar uma abordagem baseada em *Algoritmo Genético* (GA) para combinar diferentes valores de propensão a defeitos: tais valores são computados para os elementos de código a partir de heurísticas existentes. Assim, foi possível avaliar a performance da abordagem proposta, comparando valores obtidos entre: (i) métodos determinísticos: heurísticas baseadas em fluxo de controle (H) e heurísticas baseadas em fluxo de dados ($\widehat{H}$); e (ii) métodos evolucionários implementados por *Algoritmo Genético*: baseados exclusivamente em fluxo de controle (*GA-cf*) ou fluxo de dados (*GA-df*), e ainda uma abordagem híbrida (*GA_hyb*) que é baseada em ambos os fluxos. A última – *GA_hyb* – se destacou em comparação com os demais métodos, localizando mais defeitos e demandando menos elementos investigados para isso.

Na segunda abordagem proposta, objetivou-se à geração de equações completamente novas, a partir de variáveis de cobertura que expressem o espectro de fluxo de dados. Uma estratégia foi introduzida para abstrair oito novas variáveis pertinentes às coberturas de locais de definição e locais de uso das associações def-uso. A meta-heurística

Programação Genética (GP) foi utilizada para "encontrar" equações mais adaptadas ao problema de localização de defeitos. Na avaliação preliminar da abordagem proposta, os resultaram demonstraram alinhamento com os obtidos na Abordagem GA, ou seja, a abordagem híbrida, que emprega concomitantemente informação dos fluxos de controle e de dados, foi competitiva.

Ambas abordagens apresentadas foram analisadas quanto à eficácia absoluta, pela aplicação de métricas de avaliação de eficácia comumente empregadas na literatura. Além disso a abordagem GP teve uma investigação quanto à eficiência, pautada em medir o número de gerações de evolução para se obter (treinar) soluções. Nesse sentido, a abordagem híbrida (*GP_hyb*) obteve um número reduzido de gerações para expressar a mesma eficácia que GPs baseadas exclusivamente em fluxo de controle ou fluxo de dados. Em adição, *GP_hyb* alcança melhores resultados conforme as gerações passam, enquanto os outros métodos "ficam estagnados" após alguns ciclos de execução.

Dos resultados descritos acima, ressalta-se a importância da análise de fluxo de dados como fonte de informação sobre defeitos no software. Os números demonstram a boa adaptação do espectro de fluxo de dados a abordagens evolucionárias, bem como a complementariedade de informação dos fluxos de controle e de dados na evolução de métodos competitivos para a localização de defeitos. Essas conclusões foram inicialmente sustentadas pela análise estatística preliminar sobre os experimentos aplicados.

Ao utilizar a combinação de duas fontes de informações nas abordagens híbridas, é possível utilizar boas características de cada fonte. Informação de fluxo de controle pode "filtrar" alguns elementos, e fluxo de dados ser utilizado para trazer uma propensão mais granular do defeito nesses elementos "filtrados". Fluxo de controle tende a apontar locais de uso de variáveis, mostrando onde o defeito é percebido. Fluxo de dados por sua vez tem capacidade de apontar o local de definição destas variáveis, em alguns casos isso já é o suficiente para localizar os defeitos. Além, disso, ao adicionar novas fontes de informação na obtenção de propensão a defeitos, impactamos diretamente nos casos de empate. Assim, conseguimos diferenciar melhor diferentes comandos, essa característica também é refletida na eficácia das abordagens híbridas.

Além das abordagens apresentas, o presente estudo inova em outros aspectos. Foi introduzida a métrica *Empate Crítico Absoluto* ($actie@n$) que busca medir *o quão dependente de estratégias de desempate os métodos de localização de defeitos são*, considerando um número limitado de comandos inspecionados. Trata-se de uma métrica que, por vocação, deve ser utilizada em conjunto com medições de eficácia.

Como trabalhos futuros pretendemos expandir os experimentos de ambas abordagens, aplicando-as em *benchmarks* conhecidamente mais robustos, como por exemplo o Defects4J [20] que é composto por diversos programas Java, oferecendo centenas de defeitos reais em programas grandes.

Este trabalho se concentrou na investigação de fluxo de dados, mas os bons resultados apresentados pelas abordagens abrem margem para outras conjecturas. Uma delas é sobre a utilização de mais fontes alternativas de informação sobre defeitos; combinar análise estática e análise dinâmica de software, utilizando abordagens evolucionárias, poderia dar início a novas análises e contribuir positivamente para futuras soluções para o problema de localização de defeitos.

Referências Bibliográficas

[1] ABREU, R.; ZOETEWEIJ, P.; C. VAN GEMUND, A. J. **An evaluation of similarity coefficients for software fault localization**. In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, p. 39–46, Dec 2006.

[2] ABREU, R.; ZOETEWEIJ, P.; VAN GEMUND, A. J. C. **On the accuracy of spectrum-based fault localization**. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, p. 89–98, Sep. 2007.

[3] ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. V. **Spectrum-based multiple fault localization**. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, p. 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[4] ABREU, R.; ZOETEWEIJ, P.; GOLSTEIJN, R.; VAN GEMUND, A. J. **A practical evaluation of spectrum-based fault localization**. *Journal of Systems and Software*, 82(11):1780 – 1792, 2009. SI: TAIC PART 2007 and MUTATION 2007.

[5] AMMANN, P.; OFFUTT, J. **Introduction to software testing**. Cambridge University Press, Cambridge, United Kingdom ; New York, NY, USA, edition 2 edition, 2017.

[6] ARCURI, A.; BRIAND, L. **A practical guide for using statistical tests to assess randomized algorithms in software engineering**. In: *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, p. 1–10, New York, NY, USA, 2011. ACM.

[7] B. LE, T.-D.; LO, D.; LE GOUES, C.; GRUNSKE, L. **A learning-to-rank based fault localization approach using likely invariants**. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, p. 177–188, New York, NY, USA, 2016. ACM.

[8] CHAIM, M. L.; MALDONADO, J. C.; JINO, M. **Poke-tool - uma ferramenta para suporte ao teste estrutural de programas baseado em analise de fluxo de dados**. In: *Simposio Brasileiro de Engenharia de Software*. Sbc, 1991.

[9] DALLMEIER, V.; LINDIG, C.; ZELLER, A. **Lightweight bug localization with ample**. In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, p. 99–104, New York, NY, USA, 2005. ACM.

[10] DE RAINVILLE, F.-M.; FORTIN, F.-A.; GARDNER, M.-A.; PARIZEAU, M.; GAGNÉ, C. **Deap: A python framework for evolutionary algorithms**. In: *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, p. 85–92, New York, NY, USA, 2012. ACM.

[11] DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Elsevier Editora Ltda., 2007. OCLC: 900850767.

[12] EIBEN, A. E.; SMITH, J. E. **Introduction to evolutionary computing**. Natural computing series. Springer, Heidelberg, 2. ed edition, 2015. OCLC: 934627991.

[13] FRASER, G.; ROJAS, J. M. **Software Testing**, p. 123–192. Springer International Publishing, Cham, 2019.

[14] GENDREAU, M.; POTVIN, J.-Y. **Handbook of Metaheuristics**. Springer, 2019. OCLC: 1057682010.

[15] HARMAN, M.; MANSOURI, S. A.; ZHANG, Y. **Search-based software engineering: Trends, techniques and applications**. *ACM Comput. Surv.*, 45(1):11:1–11:61, Dec. 2012.

[16] HOLLAND, J. H. **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. University of Michigan Press, Ann Arbor, 1975.

[17] HUTCHINS, M.; FOSTER, H.; GORADIA, T.; OSTRAND, T. **Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria**. In: *Proceedings of 16th International Conference on Software Engineering*, p. 191–200, May 1994.

[18] JANSSEN, T.; ABREU, R.; VAN GEMUND, A. J. **Zoltar: A spectrum-based fault localization tool**. In: *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime*, SINTER '09, p. 23–30, New York, NY, USA, 2009. ACM.

[19] JONES, J. A.; HARROLD, M. J.; STASKO, J. T. **Visualization for Fault Localization**. In: *in Proceedings of ICSE 2001 Workshop on Software Visualization*, p. 71–75, 2009.

[20] JUST, R.; JALALI, D.; ERNST, M. D. **Defects4j: A database of existing faults to enable controlled testing studies for java programs**. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, p. 437–440, New York, NY, USA, 2014. ACM.

[21] KOZA, J. R. **Genetic programming: on the programming of computers by means of natural selection**. Complex adaptive systems. MIT Press, Cambridge, Mass, 1992.

[22] LEITAO-JUNIOR, P. S.; FREITAS, D. M.; VERGILIO, S. R.; CAMILO-JUNIOR, C. G.; HARRISON, R. **Search-based fault localisation: A systematic mapping study**. *Information and Software Technology*, 123:106295, 2020.

[23] LYU, M. R.; HORGAN, J. R.; LONDON, S. **A coverage analysis tool for the effectiveness of software testing**. In: *Proceedings of 1993 IEEE International Symposium on Software Reliability Engineering*, p. 25–34, Nov 1993.

[24] MASRI, W. **Fault localization based on information flow coverage**. *Software Testing, Verification and Reliability*, 20(2):121–147, 2010.

[25] NAISH, L.; LEE, H. J.; RAMAMOHANARAO, K. **Spectral debugging with weights and incremental ranking**. In: *2009 16th Asia-Pacific Software Engineering Conference*, p. 168–175, Dec 2009.

[26] NAISH, L.; LEE, H. J.; RAMAMOHANARAO, K. **A model for spectra-based software diagnosis**. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, Aug. 2011.

[27] PARNIN, C.; ORSO, A. **Are automated debugging techniques actually helping programmers?** In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, p. 199–209, New York, NY, USA, 2011. ACM.

[28] PEARSON, S.; CAMPOS, J.; JUST, R.; FRASER, G.; ABREU, R.; ERNST, M. D.; PANG, D.; KELLER, B. **Evaluating and improving fault localization**. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, p. 609–620, May 2017.

[29] RIBEIRO, H. L.; DE SOUZA, H. A.; DE ARAUJO, R. P. A.; CHAIM, M. L.; KON, F. **Jaguar: A spectrum-based fault localization tool for real-world software**. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, p. 404–409, April 2018.

[30] RIBEIRO, H. L.; DE SOUZA, H. A.; DE ARAUJO, R. P. A.; CHAIM, M. L.; KON, F. **Evaluating data-flow coverage in spectrum-based fault localization**. In:

Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '19, 2019.

[31] SANTELICES, R.; JONES, J. A.; YANBING YU.; HARROLD, M. J. **Lightweight fault-localization using multiple coverage types**. In: *2009 IEEE 31st International Conference on Software Engineering*, p. 56–66, May 2009.

[32] SHAOWEI WANG.; LO, D.; JIANG, L.; LUCIA.; LAU, H. C. **Search-based fault localization**. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, p. 556–559, Lawrence, KS, USA, Nov. 2011. IEEE.

[33] SILVA-JUNIOR, D.; LEITAO-JUNIOR, P. S.; DANTAS, A.; CAMILO-JUNIOR, C. G.; HARRISON, R. **Data-flow-based evolutionary fault localization**. In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, p. 1963–1970, New York, NY, USA, 2020. Association for Computing Machinery.

[34] SOCIETY, I. C.; BOURQUE, P.; FAIRLEY, R. E. **Guide to the Software Engineering Body of Knowledge SWEBOK Version 3.0**. IEEE Computer Society Press, Los Alamitos, CA, USA, 2014.

[35] SOHN, J.; YOO, S. **Fluccs: Using code and change metrics to improve fault localization**. In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, p. 273–283, New York, NY, USA, 2017. ACM.

[36] WONG, W. E.; DEBROY, V.; GAO, R.; LI, Y. **The dstar method for effective software fault localization**. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014.

[37] WONG, W. E.; QI, Y.; ZHAO, L.; CAI, K. **Effective fault localization using code coverage**. In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, p. 449–456, July 2007.

[38] WONG, W.; GAO, R.; LI, Y.; ABREU, R.; WOTAWA, F. **A survey on software fault localization**. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[39] XU, X.; DEBROY, V.; ERIC WONG, W.; GUO, D. **Ties within fault localization rankings: Exposing and addressing the problem**. *International Journal of Software Engineering and Knowledge Engineering*, 21(06):803–827, 2011.

[40] YOO, S. **Evolving human competitive spectra-based fault localisation techniques**. In: Fraser, G.; Teixeira de Souza, J., editors, *Search Based Software Engineering*, p. 244–258, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] ZAKARI, A.; LEE, S. P.; ALAM, K. A.; AHMAD, R. **Software fault localisation: a systematic mapping study**. *IET Software*, 13(1):60–74, 2019.

[42] ZHENG, Y.; WANG, Z.; FAN, X.; CHEN, X.; YANG, Z. **Localizing multiple software faults based on evolution algorithm**. *Journal of Systems and Software*, 139:107 – 123, 2018.

APÊNDICE A

Exemplo de código instrumentado

Código A.1 Exemplo de Código Instrumentado.

```
1  void mid(int x, int y, int z) {
2    _i_function("mid");
3    _i_node("0");
4    _i_def(x, "x", "0");
5    _i_def(y, "y", "0");
6    _i_def(z, "z", "0");
7    _i_node("1");
8    int m = _i_cuse(x, "x", "1");
9    _i_def(m, "m", "1");
10   if (_i_node("2") && (_i_puse(y, "y") < _i_puse(z, "z"))) {
11     _i_loc("2", "3");
12     if (_i_node("3") && (_i_puse(x, "x") < _i_puse(y, "y"))) {
13       _i_loc("3", "4");
14       _i_node("4");
15       m = _i_cuse(y, "y", "4");
16       _i_def(m, "m", "4");
17     } else {
18       _i_loc("3", "5");
19       if (_i_node("5") && (_i_puse(x, "x") < _i_puse(z, "z"))) {
20         _i_loc("5", "6");
21         _i_node("6");
22         m = _i_cuse(x, "x", "6");
23         _i_def(m, "m", "6");
24       }
25     }
26   } else {
27     _i_loc("2", "7");
28     if (_i_node("7") && (_i_puse(x, "x") > _i_puse(y, "y"))) {
29       _i_loc("7", "8");
30       _i_node("8");
31       m = _i_cuse(y, "y", "8");
32       _i_def(m, "m", "8");
33     } else {
34       _i_loc("7", "9");
35       if (_i_node("9") && (_i_puse(x, "x") > _i_puse(z, "z"))) {
36         _i_loc("9", "10");
37         _i_node("10");
38         m = _i_cuse(x, "x", "10");
39         _i_def(m, "m", "10");
40       }
41     }
42   }
43   _i_node("11");
44   printf("%d\n", _i_cuse(m, "m", "11"));
45 }
```