UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIERRY PIERRE DUTOIT

Criação Automática de Testes em Python para Plataformas de Aprendizado Com o Uso de Critérios de Cobertura Baseados em Grafos

RIO DE JANEIRO
2024

THIERRY PIERRE DUTOIT

Criação Automática de Testes em Python para Plataformas de Aprendizado Com o Uso de Critérios de Cobertura Baseados em Grafos

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Anamaria Martins Moreira
Co-orientador: Prof. Hugo Musso Gualandi

RIO DE JANEIRO
2024

CIP - Catalogação na Publicação

D977c

Dutoit, Thierry Pierre
Criação Automática de Testes em Python para Plataformas de Aprendizado Com o Uso de Critérios de Cobertura Baseados em Grafos / Thierry Pierre Dutoit. -- Rio de Janeiro, 2024.
72 f.

Orientadora: Anamaria Martins Moreira.
Coorientador: Hugo Musso Gualandi.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. correção automática de programas. 2. projeto de testes. 3. cobertura de grafos. 4. inteligência artificial. 5. python. I. Moreira, Anamaria Martins, orient. II. Gualandi, Hugo Musso, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

THIERRY PIERRE DUTOIT

Criação Automática de Testes em Python para Plataformas de Aprendizado Com o Uso de Critérios de Cobertura Baseados em Grafos

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 25 de março de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
ANAMARIA MARTINS MOREIRA
Data: 02/04/2024 09:29:27-0300
Verifique em https://validar.iti.gov.br

Anamaria Martins Moreira
Doutora (IC/UFRJ)

gov.br Documento assinado digitalmente
HUGO MUSSO GUALANDI
Data: 02/04/2024 09:45:40-0300
Verifique em https://validar.iti.gov.br

Hugo Musso Gualandi
Doutor (IC/UFRJ)

gov.br Documento assinado digitalmente
HUGO DE HOLANDA CUNHA NOBREGA
Data: 02/04/2024 10:27:06-0300
Verifique em https://validar.iti.gov.br

Hugo de Holanda Cunha Nobrega
Doutor (IC/UFRJ)

gov.br Documento assinado digitalmente
CARLA AMOR DIVINO MOREIRA DELGADO
Data: 02/04/2024 10:48:26-0300
Verifique em https://validar.iti.gov.br

Carla Moreira Delgado
Doutora (IC/UFRJ)

À minha família que me apoiou durante a graduação. Minha mãe Francione por todo o amor, apoio e sacrifícios que você fez ao longos dos anos em prol dos meus estudos. Meu pai Eric, que viabilizou financeiramente a minha graduação e meus dois irmãos, que compartilharam o processo de graduação comigo. À meus amigos e professores com quem compartilhei essa jornada na UFRJ, em especial meu amigo Vinícius Lettiéri, por ser minha fonte constante de inspiração e por sempre me ajudar.

AGRADECIMENTOS

Gostaria de agradecer à orientadora deste projeto, Anamaria Martins Moreira, pelo acompanhamento recorrente e sugestões que guiaram o trabalho. Agradeço também ao Hugo Musso Gualandi, pela troca de conhecimento, sugestões e enorme apoio na revisão deste projeto. Agradeço o Hugo Nobrega por ter me orientado durante a graduação e ter me incentivado a permanecer na faculdade durante todo este tempo. Agradeço também o Phillip Schanely, o principal contribuidor do CrossHair, que mostrou-se disponível durante o desenvolvimento deste projeto com quaisquer dúvidas relacionadas ao CrossHair. Por último, agradeço minha família, cujo apoio viabilizou e incentivou a conclusão deste trabalho, e à minha namorada, Beatriz, que se mostrou paciente e atenciosa durante essa etapa.

"Program testing can be used to show the presence of bugs,
but never to show their absence."

Edsger Dijkstra

RESUMO

Os testes de unidade são parte fundamental da garantia de qualidade de software, pois visam validar módulos individuais de código, como funções, métodos ou classes, reduzindo os riscos de falha. No entanto, o processo de criação de testes demanda bastante tempo, seja pela complexidade do software, necessidade de manutenção contínua, dependências externas, ou outros empecilhos. Como consequência, atingir a cobertura desejada de testes nem sempre é alcançável, expondo o software a riscos. Uma alternativa, é automatizar a criação dos testes, garantindo maior eficiência do tempo das pessoas e diminuindo custos. Para guiar essa automação utilizamos critérios de cobertura de testes baseados em grafos e, através de técnicas como execução concólica e fuzzing, conseguimos criar automaticamente os testes de unidade. Visto que essas técnicas são descontextualizadas e não levam em consideração as nuances do programa, eventualmente valores não muito intuitivos são usados como entrada dos testes gerados. Para solucionar isso usamos um Large Language Model (LLM), em específico o ChatGPT, que através do enunciado do problema em linguagem natural, retorna entradas mais intuitivas. Para avaliar a nossa ferramenta, usamos a plataforma de aprendizado Machine Teaching. Ela é uma plataforma de aprendizado de Python cuja correção dos problemas é feita de forma automática, a partir de testes criados pelos docentes. A disponibilização da nossa ferramenta visa contribuir na criação de novos problemas na plataforma Machine Teaching, assim como na validação dos problemas já existentes, diminuindo o trabalho manual dos professores e o tempo total gasto na criação de testes. A partir dos experimentos realizados, verificamos que os testes criados pela nossa ferramenta aumentam a cobertura de testes da Machine Teaching. Além disso, validamos a viabilidade do seu uso dado o seu curto tempo de execução e baixo custo monetário associado às requisições ao ChatGPT.

Palavras-chave: correção automática de programas; projeto de testes; cobertura de grafos; inteligência artificial; python.

ABSTRACT

Unit tests are a fundamental part of software quality assurance, as they aim to validate individual code modules, such as functions, methods, or classes, reducing the risks of failure. However, the process of creating tests demands a lot of time, whether due to the complexity of the software, the need for continuous maintenance, external dependencies, or other obstacles. As a consequence, achieving the desired test coverage is not always attainable, exposing the software to risks. An alternative is to automate the creation of tests, ensuring greater efficiency of people's time and reducing costs. To guide this automation, we use test coverage criteria based on graphs and, through techniques such as symbolic execution and fuzzing, we can automatically create unit tests. Since these techniques are decontextualized and do not take into account the nuances of the program, values that are not very intuitive are occasionally used as inputs for the generated tests. To solve this, we use a Large Language Model (LLM), specifically ChatGPT, which, through the statement of the problem in natural language, returns more intuitive inputs. To evaluate our tool, we use the Machine Teaching learning platform. It is a Python learning platform whose problem correction is done automatically, based on tests created by the instructors. Our tool aims to contribute to the creation of new problems on the Machine Teaching platform, as well as to the validation of existing problems, reducing the manual work of teachers and the total time spent on test creation. From the experiments conducted, we verified that the tests created by our tool increase the test coverage of Machine Teaching. Additionally, we validated the feasibility of its use given its short execution time and low monetary cost associated with requests to ChatGPT.

Keywords: automatic program correction; test project; graph coverage; artificial intelligence; python.

LISTA DE ILUSTRAÇÕES

Figura 1 – Grafo de Fluxo de Controle freq palavras 20
Figura 2 – Diferença cobertura de nós e arestas 22
Figura 3 – Exemplo de Fuzzing para string 25
Figura 4 – Comandos Executados um Teste do Problema de Frequência de Palavras 43
Figura 5 – Grafo de Fluxo de Controle freq palavras 45
Figura 6 – Testes Filtrados do CrossHair 46
Figura 7 – Testes do ChatGPT 46
Figura 8 – Resultado Final 47
Figura 9 – Custo dos Modelos do GPT Utilizados 48
Figura 10 – Custo dos Modelos do GPT Utilizados 48
Figura 11 – Tempo de Execução CrossHair (em segundos) 55
Figura 12 – Tempo de Execução ChatGPT (em segundos) 55
Figura 13 – Tempo ChatGPT Sem problemas 835 e 836 (em segundos) 56
Figura 14 – Testes Machine Teaching Problema 800 65
Figura 15 – Testes CrossHair Problema 800 66
Figura 16 – Testes ChatGPT Problema 800 66
Figura 17 – Testes Finais Problema 800 67
Figura 18 – Testes Machine Teaching Problema 828 67
Figura 19 – Testes CrossHair Problema 828 68
Figura 20 – Testes Finais Problema 828 68
Figura 21 – Testes CrossHair e ChatGPT Problema 742 69
Figura 22 – Testes Finais Problema 742 70
Figura 23 – Testes CrossHair e ChatGPT Problema 744 70
Figura 24 – Testes Finais Problema 744 71

LISTA DE CÓDIGOS

Código 1 Metódo em Python para contar a frequência de palavras em uma frase 19
Código 2 Código calcula média de idade 23
Código 3 Condição com valor simbólico 27
Código 4 Testes gerados pelo CrossHair para o método freq palavras 29
Código 5 Erro de Atributo encontrado ao gerar testes para o método freq palavras 29
Código 6 Método freq palavras com anotações de tipo 30
Código 7 `Testes do crosshair para o método freq_palavras` 31
Código 8 `Testes com uso do ChatGPT para o método freq_palavras` . . 32
Código 9 `Comando de Execução do CrossHair` 39
Código 10 `Comando de Execução do ChatGPT` 41
Código 11 `Testes CrossHair` 45

LISTA DE QUADROS

Quadro 1 – Comparativo de Cobertura (%) Testes Atuais e Novos - Todos os Critério 52
Quadro 2 – Requisitos Satisfeitos Critério de Par de Arestas: CrossHair e ChatGPT 54

LISTA DE ABREVIATURAS E SIGLAS

LLM	Large Language Model
API	Application Programming Interface
TR	Test Requirement
MT	Machine Teaching
GPT	Generative Pre-trained Transformer
UFRJ	Universidade Federal do Rio de Janeiro

SUMÁRIO

1 INTRODUÇÃO 14
1.1 CONTEXTO E MOTIVAÇÃO 14
1.2 OBJETIVO 15
1.3 CONTRIBUIÇÃO 15
1.4 ESTRUTURA DO TRABALHO 16

2 CONCEITOS BÁSICOS 17
2.1 CRITÉRIO DE COBERTURA DE TESTES BASEADO EM GRAFOS 18
2.1.1 Critérios de Nós 21
2.1.2 Critérios de Arestas 21
2.1.3 Critérios de Pares de Arestas 21
2.2 EXECUÇÃO SIMBÓLICA 22
2.2.1 Fuzzing 25
2.2.2 Execução Concólica 26
2.2.3 CrossHair 27
2.3 LARGE LANGUAGE MODEL E CHATGPT 30

3 PROPOSTA DO TRABALHO 33
3.1 PROPOSTA INICIAL E OBJETIVOS 33
3.2 CONTEXTO DO TRABALHO 34
3.3 METODOLOGIA E PROCESSO DE TESTE 35

4 IMPLEMENTAÇÃO DA NOSSA FERRAMENTA 37
4.1 PLANEJAMENTO DA CRIAÇÃO DA FERRAMENTA 37
4.2 CÓDIGO PRINCIPAL E ARQUIVOS AUXILIARES 38
4.2.1 Código Principal 38
4.2.2 Comando de Execução do CrossHair 39
4.2.3 Comando de Execução do ChatGPT 40
4.2.4 Funções Auxiliares 42
4.3 EXEMPLO 44
4.3.1 Problema de contagem de palavras em uma frase 44
4.4 TEMPO DE EXECUÇÃO E CUSTOS 47

5 ANÁLISE DOS RESULTADOS 49
5.1 OBJETIVOS 49
5.2 EXPERIMENTOS REALIZADOS 50

5.3 ANÁLISE DOS RESULTADOS 50
5.4 TEMPO DE EXECUÇÃO E CUSTOS 54
5.4.1 Tempo de Execução 54
5.4.2 Custo do GPT 56
5.5 LIMITAÇÕES ENCONTRADAS DURANTE O PROCESSO DE TESTE 57

6 CONCLUSÃO 59
6.1 TRABALHOS FUTUROS 59

REFERÊNCIAS 61

ANEXO A – PROMPT DA REQUISIÇÃO AO CHATGPT . 64

ANEXO B – EXEMPLOS ONDE A NOSSA FERRAMENTA OBTEVE COBERTURA MAIOR QUE A MACHINE TEACHING 65
B.0.1 Exemplo 1 65
B.0.2 Exemplo 2 66

ANEXO C – EXEMPLOS DE SUBSTITUIÇÃO DAS ENTRADAS PELO LLM 69
C.0.1 Exemplo 1 69
C.0.2 Exemplo 2 70

ANEXO D – LINKS PARA MATERIAL DO PROJETO . . 72

1 INTRODUÇÃO

1.1 CONTEXTO E MOTIVAÇÃO

Os testes de unidade são parte fundamental da garantia de qualidade de software, pois visam validar módulos individuais de código, como funções, métodos ou classes, reduzindo os riscos de falha. No entanto, o processo de criação de testes demanda bastante tempo, seja pela complexidade do software, pelo baixo número de testes em sistemas legados, necessidade de manutenção contínua, dependências externas, ou outros empecilhos (BARR et al., 2014). Como consequência, atingir a cobertura desejada de testes nem sempre é alcançável, expondo o software a riscos.

Uma alternativa, que pode vir a ser mais eficiente, é automatizar a criação de testes de unidade, garantindo maior eficiência do tempo das pessoas, diminuindo custos, aumentando a segurança e até auxiliando na pontualidade da entrega do software (AMMANN; OFFUTT, 2017). Para guiar essa automação podem ser utilizados critérios de cobertura de testes baseados em grafos. A partir de um critério de cobertura temos requisitos, representando caminhos diferentes a serem percorridos no módulo em teste. Quanto mais caminhos forem testados mais seguro o software estará.

Para aplicar a automação, usamos a plataforma de aprendizado Machine Teaching (MORAES et al., 2022), dada a compatibilidade e a necessidade de testes para avaliar os programas dos alunos. Ela é uma plataforma de aprendizado em Python focada em alunos que estão iniciando os estudos em programação. Na plataforma, a correção dos problemas é feita de forma automática, a partir de testes criados pelos docentes. Com isso, ter algum mecanismo de automação do projeto de testes para integrar à Machine Teaching pode auxiliar os professores na criação de novos problemas.

Dada a importância dos testes e o tempo gasto para criá-los, a nossa ferramenta de automação faz uso do CrossHair por garantir uma boa cobertura. Através de técnicas como execução simbólica e fuzzing, junto a um critério de cobertura, o CrossHair pode criar testes referentes à solução de um professor. Visto que o CrossHair não sabe o contexto do problema, ele eventualmente faz uso de valores confusos a serem apresentados aos alunos, como por exemplo a string `"\x00\x00\x00"` representando uma palavra. Para solucionar isso, usamos um Large Language Model(LLM), que através do enunciado do problema em linguagem natural, retorna entradas mais intuitivas em relação ao contexto do problema. Por fim, o conjunto de testes é disponibilizado no formato adequado para integração com a plataforma Machine Teaching.

O trabalho visa contribuir com a criação de uma ferramenta de geração automática de testes combinando o uso de critérios de cobertura baseados em grafos, execução concólica e Large Language Models. A disponibilização dessa ferramenta contribuirá na criação de

novos problemas na plataforma Machine Teaching, assim como na validação dos problemas já existentes, evitando o trabalho manual dos professores e diminuindo o tempo total gasto. Com esta ferramenta, o papel do professor volta-se para a curadoria dos testes, identificando se as entradas estão condizentes com o problema e se todos os cenários projetados foram explorados.

1.2 OBJETIVO

O objetivo deste trabalho é propor uma ferramenta para geração automática de testes em Python, com maior foco maior em problemas iniciais provenientes de plataformas de aprendizado. Essa geração automática fará uso de diversas técnicas e ferramentas, como execução simbólica, execução concreta, fuzzing e Large Language Model(LLM), para percorrer múltiplos caminhos de execução de um artefato de software.

Para guiar este processo serão utilizados três critérios de cobertura de testes baseado em grafos: nós, arestas e pares de arestas. A escolha dos critérios baseados em grafos é motivada pela maior facilidade em automatizar o projeto de testes e, também, pelos resultados obtidos em um estudo. Este estudo consistiu em analisar o desempenho de diversos critérios de cobertura em problemas provenientes da Machine Teaching, e o critério de cobertura baseado em grafos apresentou o melhor custo-benefício (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023). Com estes três critérios poderemos comparar a cobertura que cada um alcança e compreender qual critério é mais vantajoso dado a quantidade de testes e a complexidade associada. Analisaremos também a quantidade atual de testes para os problemas na plataforma e a quantidade obtida através da automação, verificando se a automação consegue aumentar a quantidade de caminhos diferentes explorados.

Por fim, com a ferramenta de geração automática de testes finalizada e com os resultados em mãos, analisaremos o ganho e a viabilidade de usá-la. Caso tenha um bom desempenho, ela poderá ser utilizada em plataformas de ensino de Python auxiliando os professores na geração dos casos de teste dos problemas de estudo. A ferramenta poderá ser usada tanto para validar os problemas já existentes na plataforma, quanto para ajudar na criação dos novos problemas.

1.3 CONTRIBUIÇÃO

Este trabalho contribui com o aumento da cobertura de teste de artefatos de software feitos em Python ao fornecer uma ferramenta que gera testes automaticamente. O foco inicial da nossa ferramenta é para plataformas de aprendizado, dada a compatibilidade dos problemas com a automação, mas a ferramenta pode ser utilizada em qualquer contexto que não faça uso de bibliotecas externas.

Também contribuímos para o estudo de um conjunto de técnicas e ferramentas, a partir da união do CrossHair, LLM e critérios de cobertura baseados em grafos. A combinação

dessas técnicas superou o desempenho individual de cada, ao passo elas se complementam para atingir uma maior cobertura de testes.

Além disso, o trabalho contribui para a pesquisa e aprimoramento das ferramentas open source utilizadas. Como por exemplo o CrossHair, amplamente utilizado neste trabalho, no qual alguns defeitos foram encontrados e reportados para o maior contribuinte da ferramenta.

1.4 ESTRUTURA DO TRABALHO

O restante deste trabalho é dividido em 5 capítulos. O capítulo 2 introduz conceitos necessários para a compreensão do trabalho. Inicialmente, é passada a teoria básica de teste de software, como termos e como conceitos básicos, critérios de cobertura e, em mais detalhes, os critérios de cobertura de testes baseados em grafos que são usados neste trabalho. Em seguida, são introduzidas as técnicas utilizadas na ferramenta deste trabalho, como execução simbólica e fuzzing. Por fim, são apresentadas as ferramentas utilizadas, explicando inicialmente o CrossHair e em seguida o uso de LLM, focando especialmente no ChatGPT.

O capítulo 3 apresenta em mais detalhes a proposta do trabalho, assim como os objetivos e metodologia utilizada. Este capítulo também menciona o contexto e o escopo do trabalho e, mais para o final, o processo de teste realizado.

O capítulo 4 foca em explicar a implementação da nossa ferramenta de geração automática de testes. De início temos o planejamento da ferramenta e a configuração para usá-la. Em seguida, há uma explicação detalhada do funcionamento da nossa ferramenta, um exemplo para auxiliar na compreensão e os custos e tempos de execução associados ao uso da ferramenta.

O capítulo 5 faz a análise dos testes sobre a nossa ferramenta. De início dizemos os objetivos dos testes para verificarmos no final do capítulo se foram ou não atingidos. Também analisamos, através de um conjunto de problemas, o desempenho de ferramenta em relação aos testes existentes no Machine Teaching. Por fim, analisamos o tempo de execução e o custo monetário associado à nossa ferramenta, indicando a viabilidade do seu uso e as suas limitações.

O capítulo 6 sintetiza todas as conclusões alcançadas ao longo do estudo e menciona áreas relacionadas e trabalhos futuros, de acordo com os resultados deste trabalho.

2 CONCEITOS BÁSICOS

No processo de criação de testes de software é crucial a definição de critérios de cobertura para metrificar a qualidade dos testes criados. Um critério como o número total de testes não é interessante pois os testes podem ser redundantes, explorando poucos ou até mesmo um único caminho de execução. Nesse contexto, os critérios de cobertura visam garantir que os testes explorem os inúmeros cenários possíveis dentro dos programas (AMMANN; OFFUTT, 2017). A utilização dos critérios diminui a redundância nos testes, impactando diretamente no tempo e no custo exigido para executá-los e mantê-los.

De acordo com Ammann e Offuut é possível condensar todos os critérios de cobertura de teste em um conjunto limitado de critérios, agrupando-os em apenas quatro estruturas matemáticas: domínios de entrada, grafos, expressões lógicas e sintaxes (AMMANN; OFFUTT, 2017). Essas estruturas representam diferentes aspectos de um sistema de software que podem ser testados, e a escolha dos critérios a serem usados depende de vários fatores, incluindo os objetivos específicos de teste, a natureza do software e os recursos disponíveis.

1. Domínios de Entrada: Este critério se concentra em testar vários valores de entrada ou intervalos de valores que um componente de software pode aceitar. É uma boa escolha quando você deseja garantir que o software lide corretamente com uma ampla gama de entradas. Além disso, não é necessário compreender a implementação do software, pois tudo se baseia na descrição dos entradas, e a técnica pode ser ajustada conforme necessário para aumentar ou diminuir o número de testes.

2. Grafos: A ideia do critério de cobertura baseado em grafos consiste em extrair um grafo a partir de um artefato de software em teste, e atravessar o grafo de alguma forma. A partir de um grafo podemos definir alguns critérios como: cobertura de nós, cobertura de arestas e cobertura de caminhos. Como exemplo, temos a abstração de um código para um grafo a partir do mapeamento das ramificações e instruções, chamado de grafo de controle de fluxo. Nesse caso, para satisfazer a cobertura de nós, todos os comandos do artefato devem ser testados. A mesma lógica se aplica para os outros critérios, porém considerando arestas ou caminhos em vez de comandos.

3. Expressões Lógicas: Esse critério é relevante quando você precisa testar as diferentes condições e expressões lógicas em seu software, como condições booleanas e pontos de decisão. Seu uso é acentuado quando o artefato de software depende de muitas condições para tomar uma decisão diferente no fluxo de execução.

4. Sintaxes: Em casos em que o artefato de software em teste pode ser descrito usando uma sintaxe como uma gramática ou uma linguagem formal, critérios de cobertura baseados em sintaxe são úteis. Por exemplo, utilizando como exemplo uma gramática, podemos ter como critérios a cobertura de símbolos terminais e a cobertura de derivações.

A escolha de quais critérios de cobertura usar depende dos objetivos específicos de teste e da natureza do software sendo testado. No contexto deste trabalho alguns critérios não são muito eficazes. Felizmente, um estudo já foi realizado comparando o desempenho entre os critérios no conjunto de problemas do Machine Teaching (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023). Neste estudo foi identificado que o critério de grafos apresenta um bom desempenho, como mencionado na sua conclusão:

> O critério de grafos, no geral, exigiu menos tempo para modelar os seus testes e também menos casos de teste por problema em relação aos outros critérios. Além disso, o critério de grafo também apresentou melhor desempenho. Considerando esses fatores, os testes projetados a partir do critério de grafo demonstraram melhor custo-benefício.

Logo, aproveitando o trabalho relacionado, o critério que melhor se adapta ao contexto da plataforma de aprendizado Machine Teaching é o critério de grafos. É um critério compatível com a ideia de automação e que conseguirá explorar os diversos caminhos dos problemas da plataforma de aprendizado.

2.1 CRITÉRIO DE COBERTURA DE TESTES BASEADO EM GRAFOS

Um artefato de software, que pode ser um simples método em um programa, pode ser representado como um grafo. Formalmente, tal grafo é composto por:

- um conjunto V de nós
- um conjunto V_0 de nós iniciais, tal que $V_0 \subseteq V$
- um conjunto V_f de nós finais, tal que $V_f \subseteq V$
- um conjunto E de arestas, ou arcos, onde E é um subconjunto de $V \times V$

Os grafos mais relevantes neste trabalho serão os grafos de fluxo de controle. Eles são uma representação visual e matemática usada para descrever o fluxo de controle de um programa ou código-fonte. Com eles é possível analisar a estrutura e o comportamento de um artefato de software, em termos de como as instruções são executadas e quais caminhos de execução podem ser percorridos.

Os nós no grafo de fluxo de controle representam uma instrução do programa. Isso pode incluir declarações, comandos de controle (como estruturas condicionais e loops), ou qualquer outra unidade de execução no programa. As arestas conectam os nós do grafo e representam a sequência lógica das instruções. Elas indicam a ordem em que as instruções são executadas. Por exemplo, uma aresta pode conectar um nó que representa a primeira instrução de um bloco condicional a outro nó que representa a próxima instrução a ser executada, dependendo da condição.

Para elucidar os conceitos vejamos um exemplo. Seja a função em Python que contabiliza a frequência das palavras em uma frase, disposto no Código 1. Como parâmetro, ele recebe uma string, e retorna um dicionário com chaves correspondendo às palavras e o valor contabilizando a frequência das palavras.

Código 1 – Metódo em Python para contar a frequência de palavras em uma frase

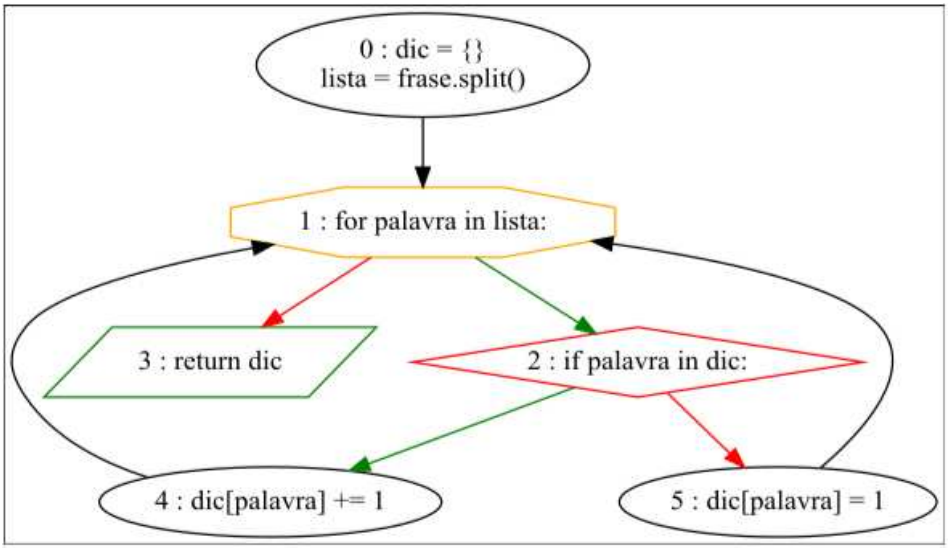
```
def freq_palavras(frase):
    dic = {}
    lista = frase.split()
    for palavra in lista:
        if palavra in dic:
            dic[palavra] += 1
        else:
            dic[palavra] = 1
    return dic
```

O grafo de controle de fluxo deste artefato de software é composto por 6 nós e 7 arestas, conforme ilustrado na Figura 1. A ampliação do conjunto de nós ocorre concomitantemente à bifurcação do controle de execução. Assim, a presença de estruturas como comandos condicionais, loops ou interrupções no fluxo do código instiga a geração de novos nós. Este fenômeno é exemplificado pelos dois primeiros comandos, responsáveis por criar um dicionário vazio e extrair a lista de palavras da frase, os quais são combinados em um único nó. Posteriormente, no âmbito do comando for, desdobram-se outros dois nós, um englobando a execução interna dos comandos e o outro destinado às instruções externas ao laço.

Ao considerarmos o grafo de fluxo de controle, torna-se mais acessível a compreensão dos critérios de cobertura. O critério de cobertura é uma regra, ou coleção de regras, que geram requisitos de teste. Por exemplo, o critério de cobertura responsável por cobrir todos os comandos cria um requisito de teste para cada comando. Por sua vez, um requisito de teste é um elemento do artefato de software no qual um teste de unidade precisa satisfazer ou cobrir. Por exemplo, para um teste satisfazer um requisito associado a um comando, o teste precisa executar o comando.

Dessa forma, a partir de um critério de cobertura, derivamos os requisitos de teste, e mediante um conjunto de testes, verificamos se o conjunto de testes foi satisfeito. Como

Figura 1 – Grafo de Fluxo de Controle freq palavras



Fonte: https://pypi.org/project/py2cfg/

alguns critérios são mais exigentes que outros, a quantidade necessária de testes varia de acordo com a exigência do critério. Formalmente, segundo Ammann e Offutt, temos:

> Definição: Dado um conjunto TR de requisitos de teste para um critério de grafo C, um conjunto de testes T satisfaz C em um grafo G se e somente se, para cada requisito de teste tr em TR, existe pelo menos um caminho de teste p em path(T) tal que p atende a tr, onde path(T) é o conjunto de caminhos do grafo executados por T. (AMMANN; OFFUTT, 2017)

Além disso, o critério de cobertura baseado em grafos pode ser insatisfazível em casos nos quais há caminhos no grafo de controle de fluxo que são inatingíveis ou impraticáveis. Isso pode ocorrer por várias razões, como condições de execução que nunca são verdadeiras, loops que nunca são percorridos, ou ramificações que nunca são alcançadas devido à lógica do programa.

Se houver partes do grafo que não podem ser exercitadas durante a execução normal do software, o critério associado a essas partes pode não ser satisfazível. Isso implica que, mesmo que você tenha definido critérios de cobertura para alcançar todos os nós, arestas ou caminhos no grafo, alguns desses elementos podem permanecer sem cobertura devido à sua inacessibilidade no código-fonte. Em geral, quando o critério é menos rigoroso (nós e arestas) e ele não é satisfeito costuma ser um problema associado ao código. Quando o critério é mais rigoroso (par de arestas e caminhos), é mais comum que algum dos requisitos não seja satisfazível.

Seja TR o conjunto de requisitos e G o grafo de fluxo de controle. Os critérios mais simples, e considerados neste trabalho, são:

- Critério de nós: TR contém todos os nós alcançáveis em G
- Critério de arestas: TR contém todos os caminhos alcançáveis de tamanho até 1 em G.
- Critério de par de arestas: TR contém todos os caminhos alcançáveis de tamanho até 2 em G.

2.1.1 Critérios de Nós

A partir do critério de nós temos os requisitos de testes, como definido mais acima, que contém todos os nós alcançáveis do grafo. Vimos no exemplo que seu grafo é composto por 6 nós, resultando em um conjunto de 6 requisitos, cada um correspondente a um dos nós. Portanto, a aplicação do critério de cobertura de nós a esse grafo leva ao $TR = \{0, 1, 2, 3, 4, 5\}$ A fim de atender a este critério, é necessário um conjunto de testes que satisfaçam todos os requisitos.

Devido à simplicidade inerente ao método de frequência de palavras, essa não é uma tarefa difícil. Uma frase como "quero um teste que seja um bom teste" já se apresenta suficiente para satisfazer todos os requisitos. Nesse contexto, o conjunto de testes final se resume a apenas um teste. Embora seja verdade que o critério pode ser satisfeito por um número maior de testes, tal redundância é sub-ótima, uma vez que o objetivo é minimizar o esforço na criação de testes para satisfazer o critério.

2.1.2 Critérios de Arestas

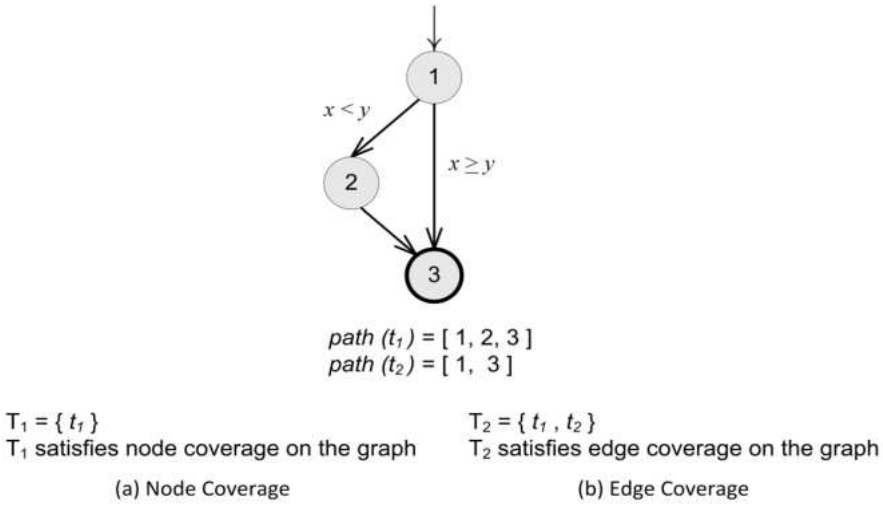
O critério de arestas segue a mesma ideia do critério de nós, apenas trocando os nós por arestas. Dessa forma, em nosso exemplo composto por 7 arestas, teremos 7 requisitos $TR = \{(0, 1), (1, 2), (1, 3), (2, 4), (2, 5), (4, 1), (5, 1)\}$. Para satisfazer o critério, precisaremos de um conjunto de teste que satisfaça todos os requisitos.

Olhando novamente a Figura 1, é fácil perceber que percorrendo todos os nós no grafo também percorremos todas as arestas. No entanto, tal evento nem sempre será verdade. Suponhamos o caso exposto na figura 2. Apenas um teste cobre todos os nós, mas dois testes são necessários para satisfazer todas as arestas. Isso ocorre por conta do if sem o else.

2.1.3 Critérios de Pares de Arestas

O critério de pares de arestas é uma extensão do critério de arestas. Em vez de considerar caminhos de tamanho até 1, consideraremos caminhos de tamanho até 2. Voltando mais uma vez para o nosso exemplo, este critério resultará no conjunto de requisitos $TR = \{(0, 1, 2), (0, 1, 3), (1, 2, 4), (1, 2, 5), (2, 4, 1), (2, 5, 1), (4, 1, 2), (4, 1, 3), (5, 1, 2), (5, 1, 3)\}$, contendo 10 requisitos. Diferente dos critérios anteriores, este é um critério mais rigoroso, e

Figura 2 – Diferença cobertura de nós e arestas



Fonte: Introduction to Software Testing - Ammann & Offutt

normalmente exige um número maior de testes para satisfazê-lo. O teste da frase "quero um teste que seja um bom teste" para este critério satisfaz 8 dos 10 requisitos, restando satisfazer $\{(0, 1, 3), (5, 1, 3)\}$. Esses requisitos remanescentes seguem caminhos mais específicos. Por exemplo, o requisito $(0, 1, 3)$, que é o caminho quando a frase é uma string vazia. O outro requisito, $(5, 1, 3)$, é um caminho que a última palavra da frase precisa ter frequência igual 1. Portanto, para satisfazer o critério de par arestas no exemplo, são necessários mais dois testes.

Vale mencionar que os critérios de cobertura de testes baseados em grafos que vimos até agora são parecidos. Se considerarmos nós como caminhos de tamanho 0, então o critério de nós é a mesma definição do critério de arestas, mas com caminhos de tamanho até 0. O critério de par de arestas é uma extensão do critério de arestas. No geral, podemos considerar: Critério de Tamanho N - TR contém todos os caminhos alcançáveis de tamanho até N em G.

Em grande parte dos casos, o critério de par de arestas possui um bom custo-benefício. Ele fornece bons caminhos de teste e não é tão oneroso quanto outros critérios mais rigorosos. De acordo com um experimento realizado por Jaffar e Lassez, o critério de par de arestas encontrou 55 defeitos com 344 casos de teste, enquanto que o critério de caminhos primos encontrou 56 defeitos com 436 casos de teste, praticamente 100 casos de testes adicionais para encontrar 1 defeito a mais (JAFFAR; LASSEZ, 1987).

2.2 EXECUÇÃO SIMBÓLICA

A ideia principal por trás da execução simbólica é utilizar valores simbólicos, em vez de valores de dados concretos como entrada, e representar os valores das variáveis

do programa como expressões simbólicas (CADAR; SEN, 2013; CLARKE, 1976; KING, 1976). Portanto, os valores de saída calculados por um programa são expressos como uma função dos valores simbólicos de entrada. Nos testes de software, a execução simbólica é empregada para gerar uma entrada de teste para um dado caminho de execução de um programa.

Para compreender melhor o processo de execução simbólica, precisamos definir alguns termos que serão usados. Os **predicados** são expressões lógicas que podem ser avaliadas como verdadeiras ou falsas, extraídos a partir dos desvios de controle de execução ao longo no artefato de software. **Símbolos** são palavras que representam valores a serem substituídos para satisfazer os predicados, eles serão gerados para cada parâmetro do programa (argumentos de funções, entradas de usuário, leitura de arquivos, etc) a partir do nome de variável associado a eles. Caso haja mais de um parâmetro associado a uma mesma variável, deve-se usar um índice para identificar cada um.

Além disso, em um artefato de software, temos três principais estruturas: **parâmetros**, **atribuições** e **predicados**. Um **parâmetro** de um programa normalmente será passado através de chamadas de função, de entradas de usuário pelo teclado, de leituras de arquivos ou de chamadas de API. Esses são os valores que irão influenciar diretamente o fluxo de execução do programa. Para cada um deles, iremos associar um símbolo. Por exemplo, na expressão em Python `soma += idade`, se a variável idade for um parâmetro iremos criar um símbolo `idade` que precisará ser resolvido depois de acordo com as restrições desejadas. Caso a expressão esteja em um loop, como no Código 2, é importante marcar cada iteração como um símbolo diferente, usando índices como `idade{i}`.

Código 2 – Código calcula média de idade

```
def media_idade(idades):
    soma = 0
    for i in range(len(idades)):
        soma += idades[i]
    return soma/len(idades)
```

As **atribuições** mudarão os valores das variáveis no programa, que precisam ser mapeados ao longo da execução. Por exemplo, para uma atribuição `ano_nascimento = ano_atual - idade`, devemos avaliar as expressões `ano_atual` e `idade`. Caso sejam símbolos, simplesmente iremos anotar o valor simbólico da variável: `ano_nascimento = ano_atual - idade`. Caso seja possível calcular `ano_nascimento`, isto é, caso os valores de `ano_atual` e `idade` sejam conhecidos, basta substituí-los e anotar o valor do resultado. Dessa forma, se `ano_atual = 2023` e `idade = 23`, denotamos `ano_nascimento: 2000`.

Durante a análise do artefato de software, precisamos guardar cada predicado de acordo com o caminho desejado para a execução do programa. Por exemplo `if (idade < 18)`, temos o predicado `idade < 18`. Caso o caminho de interesse passe pelo bloco dentro

do `if`, anotaremos o predicado `idade < 18`, caso não queiramos passar por esse bloco, anotamos o predicado negado, ou seja, `idade` $\geq$ `18`.

Para ilustrar melhor as definição podemos formalizar o processo de execução simbólica para seguir uma sequência determinada de passos. Tal algoritmo se dá por:

Entrada: Conjunto de caminhos num grafo de fluxo de controle

Saída: Conjunto de entradas que satisfaçam os requisitos de teste

Para cada caminho no conjunto de entrada faça

1. Inicialize uma lista de predicados $\boldsymbol{PC = []}$.

2. Ao identificar um parâmetro associado a uma variável **x**, inicialize um símbolo **xi**, em que **i** é o índice para identificar esse parâmetro caso esteja associado a uma variável já utilizada.

3. Comece a análise de cada comando que será executado nesse caminho.

4. Caso seja encontrada uma **atribuição** para uma variável $\boldsymbol{x = expr}$, avalie a expressão *expr*, substituindo todos os valores possíveis nela e atualize o valor da variável.

5. Caso seja encontrado um **predicado** p, substitua as variáveis dele pelos valores de cada uma. Verifique se é possível avaliar o valor verdade do predicado. Se falso, pare o método, pois indica que há uma inconsistência, ou seja, o caminho é inalcançável. Se verdadeiro, ignore esse predicado, pois ele não adiciona nenhuma restrição. Caso não seja possível avaliar o valor verdade do predicado, adicione-o à lista de predicados *PC*.

6. Retorne ao passo 4 até passar por todo o caminho.

7. Com a lista de predicado em mãos, use um *constraint solver* para encontrar valores para cada símbolo que satisfaçam todas as restrições

8. Retorne uma lista com todos os valores encontrados para cada símbolo.

O que vimos aqui é a forma mais simples de execução simbólica, mais comumente utilizada em programas estaticamente tipados. No entanto, em linguagens interpretadas como Python, esse cenário nem sempre é verdade. Além de não ser uma linguagem estaticamente tipada, que atrapalha a análise da execução simbólica, é comum lidar com estruturas de dados dinâmicas, como listas e dicionários cujo tamanho pode mudar durante a execução. Para esses problemas existem alternativas, como fuzzing e execução concólica, que auxiliam no processo de exploração dos caminhos.

2.2.1 Fuzzing

Fuzzing, a abreviação de “fuzz testing”, foi inicialmente pensada como uma técnica automática de teste que, a partir de dados inválidos ou não planejados, busca vulnerabilidades e resultados inesperados do artefato de software, em busca de mitigar a presença de vulnerabilidades exploráveis (OEHLERT, 2005). A ideia consiste em encontrar falhas em um programa através da grande quantidade de casos de teste gerados. No entanto, uma usabilidade interessante é a geração de bons casos de teste, especialmente quando guiada por alguma outra ferramenta. A síntese de valores tem auxiliado na análise de informações estáticas e dinâmicas de um programa, melhorando a produção de casos de testes e detecção de falhas (MCGRAW, 2014). Um exemplo de fuzzing para uma entrada do tipo string é apresentado na Figura 3. Gerando múltiplas entradas podemos encontrar defeitos no funcionamento do programa assim como explorar diversos fluxos de execução.

Figura 3 – Exemplo de Fuzzing para string

```
H0wJ6Fz53qtzOeQBBeDTU
MlPnQa8sJhUCSMz5nk
INKDvV8afXj
MPz6HdOLF7rnRnuCZA74XY
4
m6xbAN5sXooKMswikfuUKIs458N9BMahSI50heXv5moSJ
sqG8yRHtcvzBFS2550th3pGX4pSuMDOPY58jUR
gt29E30w
3ZwKI5jOlMpdXDpAfYp3KNLA1p32j3ShkLgoG4s6wSOfCf4Im
2bNHkyc
Qcc0Ll1
IetvoAK2HNECO0HLksEE2TFkR0XEeEww10nkiqIco7LV6fFdM
tsu44dXI8kyqZnTFQBERL
X
vu7jLN0mY9pdwoZ47vzpBzyF9e2Gb9x2xgcjXqPbtZC80
Cqqk0kyP3l6te32ABMf3QaOOhy4YWci1nhW2
YI34hRMaqUqsRMN7dlfOtwyLCIBaAZdi
CQnjP0B784fnXc7CeiQldcXDs2hes
iCXHluPiv2WEUB0Q3obsNklpGIyHyGWSUnum7
a2ytW8h18yV8fIxTUREA7qoSanggLiwH
o14UrcZRkOzoULBhDzk16yraVdEG94oRFBXzFPnn
ITAjHltKQ
4BgmwAHdjUc5qb9ldiO6vzACUvspPb2zgcA8hTNNx
```

Analisando um programa, principalmente em uma linguagem interpretada como Python, não há informação estática sobre os tipos dos parâmetros e variáveis. Nesse cenário, a técnica de fuzzing pode gerar valores indevidos, implicando em exceções durante a execução do programa. Tais exceções indicam que outros valores, ou até mesmo outros tipos, devem ser utilizados para execução completa do programa.

Nesse contexto, podemos unir a técnica de fuzzing com a técnica de execução simbólica, combinando valores simbólicos e concretos. Com isso, valores concretos são passados para um programa para ser executado simbolicamente. A união dessas duas técnicas é nomeada por execução concólica.

2.2.2 Execução Concólica

Vimos que a execução simbólica atribui valores simbólicos às entradas do programa e, ao decorrer do programa, constrói expressões de acordo com as condições booleanas encontradas. Com as condições envolvendo valores simbólicos, faz-se uso de um solucionador de restrições para obter valores concretos que satisfazem as condições encontradas.

A execução simbólica e concreta (concólica) combina fuzzing e execução simbólica para, em parte, remover as limitações da aleatoriedade do fuzzing e do teste baseado em execução simbólica: valores concretos provenientes do fuzzing são utilizados para superar parcialmente as limitações da execução simbólica, e a execução simbólica é empregada para gerar entradas de teste concretas que proporcionam uma cobertura melhor do que apenas com o uso de fuzzing (SEN, 2007). O uso de execução concólica para testar software é chamado de teste concólico, cujo objetivo é obter um conjunto de parâmetros que percorram todos os caminhos de execução viáveis (considerando um tamanho máximo para os caminhos).

A parte de execução concreta da execução concólica constitui a execução normal do programa. A parte de execução simbólica da execução concólica coleta restrições simbólicas sobre os valores simbólicos de entrada em cada ponto de bifurcação encontrado ao longo do caminho de execução. No final da execução concólica, o algoritmo obteve uma sequência de restrições simbólicas correspondentes a cada ponto de bifurcação. Vale notar que todos os valores de entrada que satisfazem as restrições de um caminho exploram o mesmo caminho de execução.

Entendendo melhor o funcionamento, o teste concólico primeiro gera valores aleatórios para entradas de tipo primitivos e valor nulo para entradas de tipo não primitivos. Em seguida, o algoritmo faz o seguinte em um loop: ele executa o código de forma concólica com a entrada gerada. No final da execução, uma restrição simbólica no conjunto de restrições do caminho é negada e resolvida usando solucionadores de restrições para gerar uma nova entrada de teste que direciona o programa ao longo de um caminho de execução diferente. O loop é repetido com a entrada de teste recém-gerada. O loop continua até que o algoritmo tenha explorado todos os caminhos de execução distintos viáveis usando uma estratégia de busca em profundidade.

Uma complicação surge do fato de que, para algumas restrições simbólicas, nosso solucionador de restrições pode não ser poderoso o suficiente para calcular valores concretos que satisfaçam as restrições. Para lidar com essa dificuldade, tais restrições simbólicas são simplificadas substituindo alguns dos valores simbólicos por valores concretos. Devido a isso, o teste concólico é completo apenas se for fornecido um oráculo que possa resolver todas as restrições em um programa, e o comprimento e o número de caminhos são finitos.

2.2.3 CrossHair

A construção de um motor de execução simbólica não é uma tarefa trivial. É claro que tal construção traria vantagens como o controle sobre o design da nossa ferramenta e a facilidade na adaptação para os problemas de um determinado contexto (como o de problemas de plataforma de aprendizado). No entanto, ao analisar a complexidade de desenvolvimento dos poucos motores de execução simbólica existentes, tornou-se claro que para o escopo deste projeto seria possível, no máximo, criar um motor básico com funcionalidades limitadas. Tais limitações poderiam acarretar em não conseguir executar o motor em grande parte dos programas em Python, distanciando este projeto do seu objetivo.

Portanto, fazendo uso do trabalho existente e evitando a construção de um motor de execução simbólica, o ponto de partida deste projeto envolveu a escolha de um motor de execução simbólica. Para linguagens não interpretadas, temos ferramentas bem difundidas como angr (CHENG, 2016) e KLEE (CADAR et al., 2008). No entanto, para linguagens interpretadas como Python, o número de ferramentas decresce consideravelmente (LUCKOW, 2023). O CrossHair se mostrou como a melhor ferramenta em Python, com mais funcionalidades e constante atualizações, para execução simbólica (Phillip Schanely, 2024). Outros projetos tiveram o mesmo dilema, também escolhendo o CrossHair como motor de execução simbólica para Python (MONTANDON, 2022). Diferentemente da execução concólica, os valores do CrossHair começam cada execução com valores puramente simbólicos e não recebem valores concretos até que sejam necessários. Basicamente o CrossHair invoca a função em teste repetidamente passando objetos especiais que se comportam como os parâmetros esperados pela função. Esses objetos especiais se comportam de forma diferente em cada execução, explorando caminhos distintos.

Esses objetos especiais possuem uma ou mais restrições que serão usadas posteriormente no solucionador de restrições. O solucionador de restrições usado no CrossHair é o Z3(MOURA; BJØRNER, 2008). Então, por exemplo, caso uma função receba como parâmetro um inteiro, o CrossHair fornecerá um objeto especial do tipo inteiro simbólico.

O parâmetro simbólico só receberá um valor concreto quando necessário. Por exemplo, considerando o Código 3, temos um valor simbólico dentro de uma expressão condicional que envolve um valor concreto. É necessário associar o valor simbólico a um valor concreto para decidir se a condição será satisfeita. A decisão do valor, nesse caso verdadeiro ou falso, é feita aleatoriamente. Em seguida, a restrição é armazenada no conjunto de restrições e o valor concreto é retornado para continuar a execução do programa.

Código 3 – Condição com valor simbólico

```
if symbolic_x > 0:
    print('bigger than zero')
```

O CrossHair lembrará quais decisões foram tomadas para que possa tomar decisões diferentes em execuções futuras. Em última análise, estamos procurando que algo específico aconteça: uma exceção ser lançada ou uma pós-condição retornar falso. Quando isso acontece, pedimos ao Z3 um modelo e o relatamos como um contraexemplo, assim explorando outros caminhos.

Um ponto interessante do CrossHair são os tipos dos parâmetros. Em linguagens dinâmicas como Python, valores de tipos diferentes podem ser atribuídos à uma mesma variável. Portanto, o CrossHair pede anotações de tipo para ajudar na escolha de valores e diminuir o tempo necessário para a análise do programa. Quando não há indicação do tipo, o CrossHair tenta aleatoriamente valores de qualquer tipo. Se ele escolher um valor de um determinado tipo e conseguir chegar até o final do programa, então ele terá um caso de teste com o tipo escolhido. Caso contrário, se durante a execução do programa receber um erro de tipo, ele saberá que precisa escolher outro tipo na próxima execução.

Uma limitação do CrossHair, consequência do uso do resolvedor de restrições Z3, é que os casos de testes obtidos podem possuir valores não intuitivos com o contexto do problema. Considerando o problema mencionado anteriormente, de contar a frequência de palavras em uma frase, podemos usar o CrossHair e ver os testes obtidos como no Código 4. É evidente que a frase "\$x00", representando o valor 0 em notação hexadecimal, não é comum no diálogo brasileiro. Por mais que seja um teste válido, é preferível evitar o uso de caracteres especiais na construção de inputs, a não ser que os envolva. Para isso, temos duas soluções: usar uma lista de strings como palavras válidas a serem utilizadas pelo CrossHair ou usar outra ferramenta para alterar os inputs não usuais por inputs condizentes com o contexto do problema. Ambas soluções serão exploradas neste trabalho, assim como sua combinação.

Outra desvantagem é a repetição de casos de teste que percorrem o mesmo caminho de execução. No Código 4 dois caminhos de execução foram testados: a frase com uma palavra e a frase vazia. No entanto, 5 testes foram produzidos, 3 deles redundantes. Na realidade, muito mais testes redundantes foram produzidos e foram omitidos. Tal redundância ocorre dada à mecânica do CrossHair. Enquanto seu tempo limite não for atingido, ele continuará executando o programa com inputs diferentes para encontrar caminhos diferentes. Nesse método em específico o CrossHair não possui informação sobre a implementação da função .split() e tenta alterar o valor da frase para alcançar novos caminhos de execução. Eventualmente ele começará a combinar caracteres, criando frases maiores e com mais palavras, explorando mais cenários. Para evitar a redundância recorremos aos critérios de cobertura, que auxiliam no filtro de testes que executam diferentes caminhos.

Código 4 – Testes gerados pelo CrossHair para o método freq palavras

```
def test_freq_palavras_0():
    assert freq_palavras("\x00") == {"\x00": 1}

def test_freq_palavras_2():
    assert freq_palavras("") == {}

def test_freq_palavras_3():
    assert freq_palavras("\x01") == {"\x01": 1}

def test_freq_palavras_4():
    assert freq_palavras("\x02") == {"\x02": 1}

def test_freq_palavras_5():
    assert freq_palavras("\x03") == {"\x03": 1}
```

Um ponto de atenção é o pulo do teste 0 para o teste 2. Esse pulo se deu pois nossa ferramenta omitiu o teste de número 1, que havia sido gerado pelo CrossHair. Como dito anteriormente, o CrossHair vai alterando os valores de entradas para explorar diferentes caminhos de execução. Quando não há qualquer indicativo sobre os tipos dos parâmetros de entrada do método, diferentes tipos são testados. Com isso, caso seja fornecido o tipo errado é possível que a execução do programa gere uma exceção, como a demonstrada no Código 5. É importante que essas execuções ocorram para que o CrossHair descubra qual é o tipo correto a ser usado nos testes. No entanto, a falta de informação sobre os tipos dos parâmetros de entrada pode ocasionar em lentidão na geração dos testes.

Código 5 – Erro de Atributo encontrado ao gerar testes para o método freq palavras

```
def test_freq_palavras_1():
    with pytest.raises(AttributeError):
        freq_palavras(0)
```

Como solução, em Python, temos os *type hints* (dicas de tipo), uma forma de indicar ao interpretador de Python quais tipos de dados são esperados em determinados lugares no código. Os type hints foram introduzidos no Python 3.5. Eles são usados principalmente em ambientes de desenvolvimento para melhorar a legibilidade do código e permitir que ferramentas forneçam sugestões e detectem potenciais erros relacionados a tipos. A notação para usá-los é bem simples, e o Código 6 apresenta o mesmo metódo em estudo, porém com as anotações de tipo: tanto no parâmetro de entrada quanto na saída.

Código 6 – Método freq palavras com anotações de tipo

```
def freq_palavras(frase: str) -> dict:
    dic = {}
    lista = frase.split()
    for palavra in lista:
        if palavra in dic:
            dic[palavra] += 1
        else:
            dic[palavra] = 1
    return dic
```

Neste exemplo, a função `freq_palavras` recebe um argumento chamado `frase` do tipo `str` (string) e retorna um valor do tipo `dict`. Os dois pontos (`:`) são usados para indicar o tipo esperado, e a seta (`->`) é usada para indicar o tipo de retorno da função. É importante notar que no Python os type hints são ignorados em tempo de execução (type erasure). Ou seja, é possível executar o código com tipos diferentes dos indicados.

2.3 LARGE LANGUAGE MODEL E CHATGPT

Large Language Models são um tipo de inteligência artificial capaz de compreender e gerar texto em larga escala. Esses modelos são treinados em grandes quantidades de texto e são capazes de entender contextos, responder a perguntas, gerar texto coerente e também são muito utilizados para tradução entre idiomas. Eles são construídos utilizando técnicas de aprendizado de máquina, como redes neurais, e são capazes de lidar com uma variedade de tarefas relacionadas à linguagem natural. Um exemplo de Large Language Model é o GPT (Generative Pre-trained Transformer). Neste trabalho, nós usamos o ChatGPT, desenvolvido pela empresa OpenAI.

O ChatGPT, é uma ferramenta de acesso público que utiliza a tecnologia do modelo de linguagem GPT (KIRMANI, 2022). É um chatbot com a capacidade de atender a diversas solicitações textuais, desde responder a perguntas simples até realizar tarefas mais avançadas, como criar cartas de agradecimento e orientar indivíduos em debates complexos sobre questões de produtividade (LIU et al., 2023). Além de suas aplicações práticas, a habilidade do ChatGPT em gerar linguagem semelhante à humana e executar tarefas complexas representa uma inovação significativa no campo do processamento de linguagem natural e inteligência artificial.

A construção do GPT passa por um processo de duas fases: pré-treinamento generativo e não supervisionado, utilizando dados não rotulados, e ajuste fino discriminativo e supervisionado para aprimorar o desempenho em tarefas específicas. Durante a etapa de pré-treinamento, o modelo aprende de maneira natural, semelhante à forma como uma pessoa assimila conhecimento em um novo ambiente, enquanto a fase de ajuste fino

envolve um refinamento mais direcionado e estruturado pelos criadores.

Com todo esse potencial, o ChatGPT soluciona bem um problema encontrado no CrossHair: os valores de entradas gerados não condizem com o contexto do problema. Este problema é uma das desvantagens do CrossHair mencionada anteriormente, apontada pelo Código 3. Para solucionar isso, podemos criar um prompt, uma forma específica de perguntar ao ChatGPT, para manter ou aumentar a quantidade de testes e ainda percorrer os diferentes caminhos de execução. Dessa forma, mantemos a vasta exploração do CrossHair ao passo que os testes possuem valores de acordo com o contexto da criação do artefato de software. Considere o Código 7, que são os testes do CrossHair para o método de frequência de palavras antes do uso do ChatGPT. Vale notar que os testes do Código 7 são diferentes do Código 3 pois o critério de cobertura de par de arestas foi utilizado para eliminar a redundância de testes.

Código 7 – `Testes do crosshair para o método freq_palavras`

```
def test_freq_palavras():
    assert freq_palavras('another another') ==
        {'another': 2}

def test_freq_palavras_405():
    assert freq_palavras('another value') ==
        {'another': 1, 'value': 1}

def test_freq_palavras_9():
    assert freq_palavras('') ==
        {}
```

Dos 10 requisitos do critério de par de arestas para o método, 9 são satisfeitos. O único par de arestas não satisfeito é o (4, 1, 2), que precisa de uma frase com ao menos três palavras, dada a necessidade de adicionar a palavra no dicionário passando pelo nó 5, em seguida aumentar o contador da palavra passando pelo nó 4 e então ter uma palavra a mais para continuar na iteração das palavras. Satisfazer 9 de 10 requisitos é uma boa cobertura, e não chegou a 100% devido à dificuldade do CrossHair de compreender funções primitivas como o método split() de uma string.

Agora, considere o Código 8, que representa o resultado obtido após utilização da inteligência artificial. O prompt enviado ao GPT possui informações como o enunciado do problema da MT, em qual é o assunto de aula (por exemplo: aula sobre loop “for”), os testes gerados pelo CrossHair que ele usará como base e a solução do problema fornecida pelo professor. Os testes retornados pelo ChatGPT satisfazem os 10 requisitos e possuem valores que condizem bem com o contexto do problema dado pela plataforma de aprendizado. Dessa forma, a usabilidade do ChatGPT pode ser estendida. Além de contribuir com melhores valores, ele pode aumentar a cobertura de testes dos problemas, como foi o

caso. Posteriormente, veremos a diferença entre o uso somente do crosshair e o uso dele aliado à inteligência artificial.

Código 8 – Testes com uso do ChatGPT para o método freq_palavras

```
def test_1():
    assert freq_palavras('dinheiro é dinheiro e vice versa') ==
        {'dinheiro': 2, 'é': 1, 'e': 1, 'vice': 1, 'versa': 1}

def test_2():
    assert freq_palavras('dinheiro é dinheiro') ==
        {'dinheiro': 2, 'é': 1}

def test_3():
    assert freq_palavras('dinheiro') ==
        {'dinheiro': 1}

def test_4():
    assert freq_palavras('') ==
        {}
```

3 PROPOSTA DO TRABALHO

Este capítulo visa apresentar em mais detalhes a proposta, os objetivos e metodologia utilizada neste trabalho. Estes tópicos foram brevemente mencionados no capítulo 1 e merecem uma explicação mais profunda. Para isso, o capítulo foi dividido em 3 seções: a seção 3.1 fala sobre a proposta do trabalho, englobando as dores que serviram como motivação do trabalho. Em seguida são expostos os objetivos do trabalho, que visam solucionar tais dores. A seção 3.2 menciona o contexto no qual o trabalho está inserido e os motivos para a tal escolha. Por fim, a seção 3.3 fala sobre a metodologia utilizada no trabalho e o processo de teste feito em cima da ferramenta proposta.

3.1 PROPOSTA INICIAL E OBJETIVOS

O uso diário de software já faz parte da vida das pessoas na sociedade atual. Com tamanha relevância, precisamos garantir a qualidade do software, validando que o seu funcionamento está alinhado com o esperado. Para isso, podemos criar testes de unidade, que validam pequenas partes de código. A partir de um bom conjunto de testes, podemos garantir múltiplos fluxos de execução e obter maior confiança no uso de software.

No entanto, o processo de criação de testes pode demandar bastante tempo (BARR et al., 2014). Fora isso, sem um processo bem definido de criação de testes, é difícil verificar se o conjunto de testes de um software está cobrindo todos os possíveis fluxos de execução.

Tais problemas são bem relevantes em vários contextos, inclusive no contexto de plataformas de aprendizados. Assim que um problema é criado na plataforma, testes são exigidos para validar as soluções dos alunos e rejeitá-las se não estiverem de acordo com a solução do professor. Se o conjunto de testes não for bem construído, soluções indesejadas podem ser aprovadas, prejudicando o processo de aprendizado.

Para resolver os pontos mencionados acima este trabalho propõe uma ferramenta para automação da criação de testes unitários de programas escritos na linguagem Python e na framework de testes pytest. Inicialmente a ferramenta será utilizada em problemas iniciais de programação provenientes de plataformas de aprendizado, mas ela pode se mostrar útil em outros contextos.

A ferramenta auxiliará na criação de novos problemas em plataformas de aprendizado, assim como na validação dos problemas já existentes. Com a otimização do tempo do professor, soluções mais robustas podem ser construídas, além do professor poder contribuir com a criação de testes que exploram caminhos inesperados. Dessa forma, os alunos precisarão criar soluções mais resilientes, melhorando o processo de aprendizado de programação.

De forma mais detalhada, a automação da criação dos testes unitários tem os seguintes objetivos:

- Otimização do tempo gasto do professor na plataforma de aprendizado para criação de novos problemas.
- Melhor aprendizado dos alunos através de um conjunto de testes que explore todos os cenários propostos pelos problemas da plataforma.
- Validação dos problemas já existentes na plataforma em busca de testes redundantes e/ou cenários não cobertos
- Contribuir para a pesquisa e aprimoramento de ferramentas open source como o CrossHair.

3.2 CONTEXTO DO TRABALHO

Este trabalho foi projetado para ser usado em plataformas de aprendizado de programação que utilizam a linguagem Python para ensino. Inicialmente com foco na plataforma de aprendizado Machine Teaching (MORAES et al., 2022), que consiste em um ambiente da UFRJ de aprendizagem online de programação utilizado desde 2018 como ferramenta de apoio em cursos introdutórios.

A escolha do contexto se basear em plataforma de aprendizado se deve a um conjunto de três fatores, que serão explicados nesta seção.

O primeiro grande motivador são as limitações do CrossHair[1]. Algumas funções, sejam externas ou internas ao Python, podem não ser bem interpretadas pelo CrossHair. Como por exemplo a função .split() do python, no qual o CrossHair continua passando strings com apenas uma palavra por não entender a funcionalidade dela. Quanto mais funções externas ou internas um método utilizar, maior a probabilidade do CrossHair não conseguir interpretar corretamente o programa. Além disso, ele não conseguirá testar funções aninhadas ou que estão dentro de classes, forçando o uso em apenas métodos simples e enxutos. Por isso o contexto de aprendizado, onde funções mais simples e que não fazem uso de bibliotecas e funções externas, é bom para o teste inicial da ferramenta.

O segundo motivo são os trabalhos anteriores realizados em cima da plataforma Machine Teaching (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023), no qual foram fornecidas as soluções dos professores para os problemas assim como uma forma de processar as respostas dos alunos para cada um dos problemas. Tais informações ajudam no processo de teste da ferramenta de geração automática assim como apoiam na comparação da performance da ferramenta quanto aos testes já existentes na plataforma.

[1] CrossHair Limitations, https://crosshair.readthedocs.io/en/latest/limitations.html

O terceiro e último motivo consiste no objetivo principal deste trabalho: ajudar os professores na criação de novos problemas na plataforma. Dessa forma, podemos contribuir com o aprendizado na área de computação e apoiar a inserção de novas pessoas nela. Com a evolução da ferramenta ela terá utilidade em mais contextos, e pode contribuir para futuros trabalhos e projetos.

3.3 METODOLOGIA E PROCESSO DE TESTE

A escolha adequada da metodologia é essencial para assegurar a validade e confiabilidade dos resultados obtidos, bem como permitir a replicação do estudo por outros pesquisadores interessados em teste de software. Ao longo desta seção, será apresentada a metodologia utilizada neste trabalho para alcançar os pontos mencionados na proposta e nos objetivos.

Para viabilizar os testes da ferramenta e verificar se os resultados estão condizentes com a proposta é preciso obter um conjunto de problemas, com suas respectivas soluções, do Machine Teaching. Esta etapa já foi realizada por um trabalho anterior (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023), e alguns problemas estão disponível em seu repositório no Github. Para aumentar a quantidade de artefatos de software em estudo, podemos obter mais problemas considerando os da plataforma do Machine Teaching que não estão disponíveis no repositório.

Com o conjunto de problemas e suas respectivas soluções, a metodologia proposta neste trabalho consiste em:

1. Adicionar o problema e a solução na lista de problemas da ferramenta. Nessa etapa, é importante extrair o tema do contexto do problema para utilizar no prompt da LLM. Ao mesmo tempo, o repositório com as soluções não contempla todos os problemas e algumas soluções estão desatualizadas, exigindo intervenção manual.

2. Execução da ferramenta com a lista de problemas. Essa etapa é a mais demorada, visto que depende do tamanho da lista de problemas. O maior ofensor ao tempo de execução da geração de testes é o CrossHair, pois de acordo com a sua configuração o tempo médio varia.

3. Obter as métricas agrupando os resultados obtidos para cada teste. Essas métricas são importantes tanto para verificar a cobertura de testes quanto para analisar os custos associados à execução.

4. Análise das métricas com respeito aos objetivos do trabalho

5. Conclusões e desempenho da ferramenta

Esse é o processo básico proposto. As principais etapas serão detalhadas nos próximos capítulos, assim como as limitações e as dificuldades encontradas na execução da metodologia proposta.

4 IMPLEMENTAÇÃO DA NOSSA FERRAMENTA

Este capítulo visa explicar a implementação da ferramenta de geração automática de testes através de 5 seções. A seção 4.1 detalha o planejamento da criação da ferramenta e suas etapas. Na seção 4.2 são expostos os pormenores de seu funcionamento abrangendo os principais métodos e arquivos auxiliares. Na seção 4.3, temos um exemplo do funcionamento da ferramenta, enaltecendo todos os pontos mencionados nas seções anteriores. Por fim, na seção 4.4, são expostos as variações dos tempos de execução e o custos atrelados à ferramenta.

4.1 PLANEJAMENTO DA CRIAÇÃO DA FERRAMENTA

A ideia inicial deste projeto era construir uma ferramenta do zero, usando execução simbólica e outras técnicas auxiliares para a geração automática dos testes. No entanto, na etapa de pesquisa de trabalhos relacionados, algumas ferramentas robustas de execução simbólica para Python foram encontradas. Em especial, o CrossHair, que atualmente possui boa base e consegue cobrir diversos programas. Com o intuito de colaborar com o trabalho existente e progredir para uma melhor geração automática de testes em linguagens interpretadas como Python, a ferramenta aqui proposta faz uso dos benefícios do CrossHair e tenta circundar suas dificuldades com o uso de ferramentas auxiliares.

Particularmente o CrossHair possui dificuldades em lidar com strings. Como ele é desprovido de contexto, é difícil montar strings que sejam entradas relevantes para os problemas das plataformas de aprendizado. No entanto, as inteligências artificiais que estão em grande avanço são excelentes para isso. Com os modelos de LLM, é possível alterar as strings que o CrossHair faz uso por outras que condizem com o contexto do problema. Ao mesmo tempo, o uso delas pode manter ou até aumentar a quantidade de requisitos satisfeitos. No entanto, o uso de LLM's não é tão simples, dada a necessidade de uma grande quantidade de dados. Nesse contexto, o ChatGPT é uma boa solução, mas que possui um custo para o uso de sua API.

A proposta da nossa ferramenta consiste em retornar testes que satisfazem o máximo de requisitos propostos pelo critério de cobertura de grafos utilizado. Para isso, 4 entradas são necessárias: o enunciado do problema em teste, o contexto no qual ele está inserido (por exemplo: problemas sobre strings), a solução do problema fornecida pelo professor e o critério de cobertura escolhido. Veremos neste trabalho três critérios de cobertura: nós, arestas, e par de arestas. Para o uso do CrossHair, apenas a solução do problema é necessária. No entanto, o uso da inteligência artificial exige mais informações para obter um melhor desempenho, requirindo o enunciado do problema e o tema no qual o problema está inserido.

4.2 CÓDIGO PRINCIPAL E ARQUIVOS AUXILIARES

Essa seção é crucial para o entendimento do funcionamento da ferramenta e as suas etapas. Primeiro será apresentado o processo básico da ferramenta. Em seguida, um modo alternativo, que traz mais informações para uma análise melhor do processo será apresentado. Similar a um modo de debugger encontrado em outras aplicações, esse modo traz mais informações dos resultados obtidos no meio do processamento da ferramenta. Essas informações serão posteriormente compiladas e utilizadas para avaliar a performance da ferramenta. Para apoiar o processo principal métodos auxiliares são usados, que estão dispostos em arquivos separados e serão brevemente explicados nesta seção. Todas essas informações também estão dispostas no readme do repositório do GitHub[1].

4.2.1 Código Principal

O método principal da ferramenta exige quatro entradas: a solução do problema, o contexto dele, o tema e o critério de cobertura. Todos eles precisam ser passados como strings para o funcionamento correto. Em mais detalhes, temos:

- Contexto do Problema: O contexto do problema ou enunciado do problema é também fornecido pelo professor ao criar o problema. Esse contexto é muito importante na etapa que utilizada LLM, visto que quanto mais informações à cerca do problema melhor será o resultado obtido. Essa informação não é utilizada pelo CrossHair, mas é essencial para o processo.

- Solução do Problema: A solução do problema é fornecida pelo professor que criou o problema da plataforma de aprendizado. Essas soluções no Machine Teaching não são disponibilizadas para os alunos. No entanto, o trabalho realizado por (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023) disponibilizou algumas soluções para as pessoas interessadas em fazer trabalhos relacionados. Para os demais problemas foi necessário construir soluções que estivessem de acordo com o problema e os testes existentes na plataforma. A solução do problema é essencial no processo de geração dos testes pois é utilizada tanto pelo CrossHair quanto pela LLM.

- Tema do Problema: O tema é similar ao contexto do problema, também fornecido pelo professor e somente utilizado pela LLM. O tema faz referência a qual conhecimento é objeto de estudo do problema. Por exemplo, se um problema envolve strings e listas, o tema pode ser manipulação com strings e listas.

- Critério de Cobertura: O critério de cobertura baseado em grafos é um entre três valores: nós, arestas ou par de arestas. Há também a possibilidade de não passar o parâmetro de critério de cobertura, no qual nenhum filtro nos testes será realizado e

[1] GitHub Repo, https://github.com/thierryxdp/TCC/releases/tag/1.0

provavelmente haverá redundância nos testes. Por fim, é possível considerar os três critérios, que consiste em executar três vezes o código principal com cada critério.

Com as entradas bem estabelecidas, o código principal, ou processo principal, consiste nas seguintes etapas:

1. Obter os testes produzidos pelo CrossHair passando como parâmetro a solução do professor do problema.

2. Uso do critério de cobertura passado como parâmetro para filtrar os testes provenientes do CrossHair, evitando a redundância.

3. Os testes do CrossHair filtrados são passados na requisição da api do ChatGPT, junto com o enunciado do problema e o seu tema. Todos esses dados são enviados dentro de um prompt fixo, que indica à inteligência artificial o que é esperado como resposta.

4. Por último, filtramos os testes do ChatGPT junto com os testes do CrossHair, combinando os resultados obtidos pelas duas etapas. Damos prioridade para os testes do ChatGPT para manter as entradas contextualizadas. Dessa forma, os testes do CrossHair que satisfazem os mesmos requisitos que os do ChatGPT são substituídos. Os testes que possuem requisitos não satisfeitos por outros são sempre considerados no resultado final.

De forma bem resumida, o processo parece bem simples. No entanto, alguns detalhes e processamentos foram omitidos. Vejamos em detalhes como cada etapa funciona e quais partes foram omitidas.

4.2.2 Comando de Execução do CrossHair

De início, vejamos como é feita a execução do CrossHair para obtenção dos testes unitários. Após toda a configuração, executar o CrossHair consiste em executar apenas um comando, como disposto no Código 9.

Código 9 – `Comando de Execução do CrossHair`

```
crosshair cover solution.py        \
    --example_output_format=pytest \
    --coverage_type=path           \
    --per_condition_timeout=100
```

Esse comando recebe alguns parâmetros que alteram o tempo de execução do comando, o formato da saída dele e até mesmo o seu funcionamento. O parâmetro `example_output_format` determina em qual formato os testes serão retornados. Visto a necessidade de executar os

testes e retornar os resultados no formato do pytest, ele é o formato mais indicado para a nossa ferramenta.

O segundo parâmetro é o `coverage_type` que determina o tipo de cobertura a ser utilizado. Nele temos duas opções: opcode e path. O opcode é similar ao critério de cobertura de arestas, com um menor tempo de execução e menor cobertura de testes. A outra opção é o path coverage, que tenta cobrir todos os caminhos de execução do programa, resultado em maior tempo de execução e maior cobertura de testes. Como o opcode é similiar ao criterio de arestas, faz sentido usá-lo nos critérios de nós e arestas. No entanto, através dos testes, percebemos que o uso do path coverage para todos os critérios faz mais sentido. O opcode não cumpriu a promessa e na maioria dos experimentos deixou de cobrir todas as arestas.

Veremos que alguns programas possuem um número infinito de caminhos (dado a presença de loops). Para isso, temos outro parâmetro, que limita o tempo do CrossHair. É possível limitar o número de iterações ou limitar o tempo de execução de uma condição ou de um caminho. A partir dos testes, o parâmetro mais interessante a ser usado é o `per_condition_timeout`, que permite o CrossHair executar durante tempo suficiente para obter uma boa quantidade de testes. O valor utilizado nos testes foi de no máximo 100 segundos para avaliar cada condição do programa. Parece muito, mas o tempo de execução médio do CrossHair foi de 1 minuto e 27 segundos, um tempo razoável para atingir uma boa cobertura de testes.

Como consequência dos parâmetros selecionados o CrossHair pode retornar testes redundantes. Além disso, ele também retorna testes que indicam o uso errado de tipos, como mencionado no capítulo 2. Esses casos de testes não são relevantes para esse trabalho e um filtro de testes mostra-se necessário para lidar com o problema. O filtro funciona para qualquer conjunto de testes que esteja no formato do pytest, e consiste em através de um critério de cobertura, remover os testes que não satisfazem nenhum requisito insatisfeito. Ordenando os testes de forma decrescente em relação à quantidade de requisitos satisfeitos, removemos a redundância de testes ao ponto que cada teste no conjunto final precisa satisfazer ao menos um requisito que os outros testes não satisfazem.

4.2.3 Comando de Execução do ChatGPT

Agora que já vimos como funciona a execução do CrossHair e o filtro de testes, vejamos a etapa opcional, porém muito importante, do uso de uma LLM para modificar os valores de entrada dos testes. O CrossHair não lida muito bem com strings por não interpretar o contexto do problema e consequentemente não saber quais strings são relevantes como entradas para o problema. Para melhorar a produção de testes nesses casos usamos uma LLM, em específico neste trabalho o ChatGPT, para aprimorar os testes do CrossHair e retornar um resultado mais condizente para os alunos da plataforma de aprendizado. Alguns exemplos da transformação das entradas estão disponíveis no Anexo C.

Para adicionar essa etapa na ferramenta é preciso fazer requisições à api do ChatGPT na OpenAI. Essas requisições possuem um baixo custo associado que pode crescer consideravelmente de acordo com a quantidade de requisições feitas. A requisição à API do ChatGPT consiste no comando disposto no Código 10. Três parâmetros são necessários para efetuar a requisição: o modelo, as mensagens e o grau de temperatura.

Código 10 – `Comando de Execução do ChatGPT`

```
import openai

def get_chatgpt_tests(prompt, model="gpt-3.5-turbo"):
    messages = [{"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=0.2,
    )

    return response.choices[0].message["content"]
```

O modelo se refere à versão do ChatGPT a ser usada. Durante este trabalho dois modelos foram utilizados: `gpt-3.5-turbo` e `gpt-4-turbo-preview`. O primeiro foi utilizado durante o início do trabalho visto que era a versão mais recente disponível. No entanto, a partir de dezembro de 2023, a segunda versão ficou disponível. Ela foi utilizada durante três dias deste trabalho e não foi considerada por mais tempo devido ao seu custo.

As mensagens são os textos enviados ao ChatGPT para que ele responda. Em processos automáticos é interessante definir um prompt a ser enviado como mensagem para fornecer o contexto e explicar o formato de resposta esperado. O Anexo A representa o prompt utilizado neste trabalho com o problema de contar palavras em uma frase. As seções do prompt que recebem parâmetros de acordo com os problemas estão sinalizadas, para melhor entendimento.

Por fim, a temperatura é um parâmetro que controla a "criatividade" ou aleatoriedade do texto gerado pelo ChatGPT. Uma temperatura mais alta (por exemplo, 0.7) resulta em respostas mais criativas, enquanto uma temperatura mais baixa (por exemplo, 0.2) torna a saída mais determinística. Na prática, a temperatura afeta a distribuição de probabilidade sobre os tokens possíveis em cada etapa do processo de geração. Uma temperatura de 0 tornaria o modelo completamente determinístico, escolhendo sempre o token mais provável. Após o uso de diferentes temperaturas, o valor ideal encontrado foi de 0.2, forçando o ChatGPT a seguir fielmente o prompt enviado mas com espaço para ajustar os valores de entrada de acordo com o contexto dos problemas.

Após a requisição ao ChatGPT uma ou várias respostas são retornadas. Atualmente consideramos apenas a primeira resposta, e extraímos os testes da resposta. Essa extração

é fácil porque dentro do prompt pedimos que os testes sejam retornados em um formato específico, como demonstrado no Anexo A. Por fim, os testes são filtrados mais uma vez, evitando novas redundâncias que possam ter sido introduzidas pela inteligência artificial.

A etapa de execução do CrossHair e do ChatGPT são as principais no processo geral e já foram detalhadas. Entre elas existem funções auxiliares que fazem a transição de uma para outra, como a função de filtro dos testes para evitar redundâncias. Além disso, existem outras funções auxiliares que apoiam essas etapas e que merecem uma breve explicação.

4.2.4 Funções Auxiliares

A principal função auxiliar, mencionada anteriormente, é a de filtrar os testes de acordo com um critério de cobertura. Esse filtro só é possível com a ordem dos nós executados para cada teste gerado. Com um conjunto de testes para um determinado problema, podemos definir o processo de filtro em algumas etapas:

- Obter grafo de controle de fluxo do programa em teste. Isso consiste em obter tanto uma visualização do grafo quanto obter o mapeamento dos comandos para os nós e arestas do grafo. A visualização do grafo serve somente para validação visual do mapeamento e no fluxo geral não é utilizada.

- Em seguida para cada teste identificamos por quais nós ele passa e qual é a ordem de execução deles.

- Com a ordem de execução dos nós é possível dizer quais requisitos o teste satisfaz. Por exemplo, se o critério de arestas for utilizado, sabemos que quando o nó 2 é executado em seguida do nó 1 o requisito da aresta (1,2) foi satisfeito.

- Por último, sabendo quais requisitos cada teste satisfaz, fazemos o seguinte: ordenamos todos os testes em ordem decrescente em relação ao número de requisitos satisfeitos. Para cada teste, se ao menos um requisito que ele satisfaz não for satisfeito por nenhum teste do conjunto de testes filtrados, o teste é adicionado ao novo conjunto. Ao final, construímos o conjunto de testes filtrados, no qual não há redundância de testes de acordo com o critério selecionado.

Obter o grafo de controle de fluxo, ainda mais com uma imagem para visualização, não é uma tarefa trivial. É preciso mapear quais comandos estão contidos em cada nós e identificar as bifurcações no fluxo de execução que indicam as arestas entre eles. Felizmente, essa lógica foi desenvolvida por (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023), e reaproveitada neste trabalho.

No entanto, não foi encontrado no trabalho mencionado acima, uma lógica para identificar a ordem dos nós executados por um teste. Algumas ferramentas em Python foram

consideradas para sanar esse problema, mas apenas indicavam se um determinado comando foi ou não executado. Em nossas buscas não foi encontrada uma ferramenta que disponibilizasse a ordem de execução de comandos de um teste.

Além disso, não parece ser um problema complexo, é relativamente fácil para problemas enxutos dizer quais comandos serão executados de acordo com determinada entrada. Como o CrossHair executa o código em teste, podemos executá-lo novamente, para cada teste, e armazenar as linhas executadas. Com o auxílio de um arquivo e uma lógica para escrever no arquivo sempre que um comando for executado, conseguimos a lista de comandos executados por um teste, como disposto na figura 3. É possível identificar o início e o fim de execução de um teste com os comandos de "enter"e "exit", respectivamente.

Figura 4 – Comandos Executados um Teste do Problema de Frequência de Palavras

```
enter: freq_palavras(frase: str) -> dict:
dic = {}
lista = frase.split()
for palavra in lista:
if palavra in dic:
else:
dic[palavra] = 1
for palavra in lista:
if palavra in dic:
dic[palavra] += 1
for palavra in lista:
return dic
exit: freq_palavras(frase: str) -> dict:
```

Como próximo passo precisamos fazer o mapeamento dos comandos executados para os nós do grafo de controle de fluxo. O objetivo é sair de uma lista de comandos ordenada de acordo com a execução para uma lista de nós. Visto que um nó pode conter um ou mais comandos, precisamos de alguma lógica que mapeie um conjunto de comandos em seu respectivo nó. Como exemplo, vamos supor que um nó qualquer de um grafo de controle de fluxo contém três comandos. Na lista ordenada de nós podemos repetir o nó para cada um dos comandos contidos dentro dele, ao passo que executar três comandos é traduzido em passar pelo mesmo nó 3 vezes seguidas. Em seguida, se removermos os nós iguais que são adjacentes e mantermos apenas um deles, teremos a ordem exata de execução dos nós do grafo.

Para clarificar a ideia, usaremos como exemplo a lista de comandos da Figura 4. De acordo com a nossa lógica, de repetir o nó na lista de nós para comandos que pertencem a um mesmo nó, obtemos a lista de execução dos nós $[0, 0, 1, 2, \dots]$. Como os comandos `dic = {}` e `lista = frase.split()` são mapeados para o nó 0, temos o nó 0 aparecendo

duas vezes na lista. Então, basta remover os nós adjacentes iguais (mantendo um deles, é claro), que teremos a ordem de execução correta dos nós: $[0, 1, 2, \dots]$.

Com a ordem de execução dos nós para cada teste, é simples identificar quais requisitos são satisfeitos. No critério de nós, a ordem não é muito relevante. No critério de arestas e par de arestas, a ordem dos nós importa e determina as arestas ou os pares de arestas satisfeitos.

Para obter a menor quantidade de testes possível, usamos um algoritmo guloso aliado à ordenação decrescente de requisitos satisfeitos por teste. Basta iterar sobre à lista de testes e adicionar ao conjunto final todo teste que satisfazer um requisito ainda não satisfeito dentro do conjunto final.

4.3 EXEMPLO

Para elucidar a implementação e os pontos acima descritos vejamos um exemplo, do problema de contagem de palavras de uma frase, para compor a base de conhecimento. Neste exemplo focaremos no critério de par de arestas por ser o mais rigoroso e com maior quantidade de testes. Além disso, vale lembrar que satisfazer o critério de par de arestas significa fazer o critério de arestas e o critério de nós.

4.3.1 Problema de contagem de palavras em uma frase

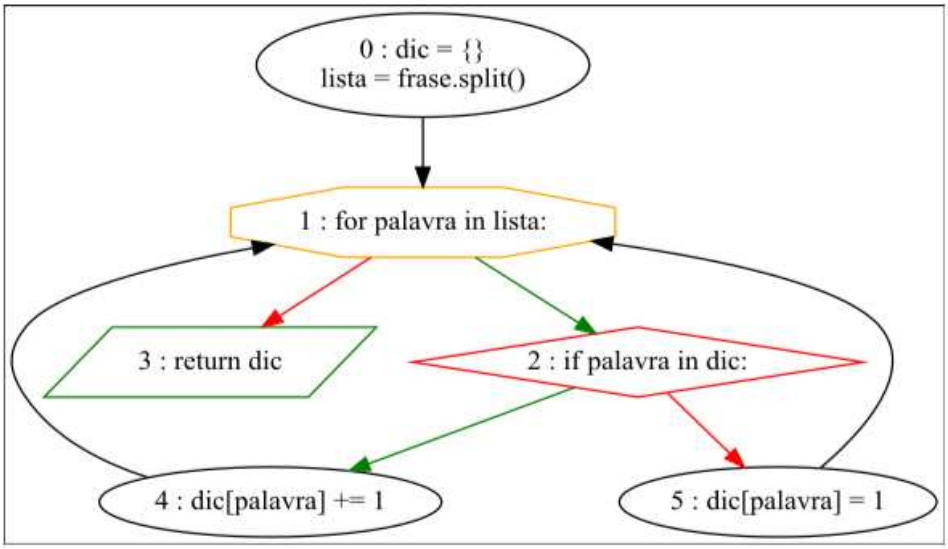
Vimos o problema de contagem de palavras em uma frase no capítulo 2 deste trabalho. Pela familiaridade com o problema e por sua solução não ser muito complexa, ele se apresenta como um forte candidato para o exemplo inicial. Relembrando o problema, ele consiste em criar uma função chamada `freq_palavras` que recebe uma string (uma frase) e retorna um dicionário com a chave representando a palavra e o valor a frequência dela na frase.

Para gerar os testes unitários para o problema basta executar o método `generate_unit_tests` passando como parâmetro a solução do problema, o enunciado e o tema. A solução do problema foi apresentada no capítulo 2, assim como uma melhor explicação sobre o enunciado e o tema. Existe um último parâmetro chamado debugger, booleano, que determina se os testes não filtrados devem aparecer em tela. Para visualizar o funcionamento da ferramenta passaremos esse parâmetro como verdadeiro, mas por padrão ele é falso.

A primeira informação apresentada em tela é o grafo de controle de fluxo da solução do problema. Esse grafo é importante para a análise dos requisitos e somente é apresentado quando o modo debugger está ativado. O grafo de fluxo de controle para o problema de contagem de palavras já foi apresentado no capítulo 2, mas está novamente representado pela Figura 5 para melhor fluidez na leitura.

Em seguida o comando de criação de testes do CrossHair é executado com a solução do professor como parâmetro. Essa etapa gera uma grande quantidade de testes redundantes,

Figura 5 – Grafo de Fluxo de Controle freq palavras



tornando inviável a apresentação de todos os testes. Portanto, apresentamos apenas os três primeiros e três últimos testes dessa etapa, como exemplificado no Código 11.

Código 11 – `Testes CrossHair`

```
def test_freq_palavras():
    assert freq_palavras('another another') == {'another': 2}

def test_freq_palavras_6():
    assert freq_palavras('') == {}

def test_freq_palavras_28():
    assert freq_palavras(' ') == {}

def test_freq_palavras_681():
    assert freq_palavras('   value') == {'value': 1}

def test_freq_palavras_689():
    assert freq_palavras('test value') == {'test': 1, 'value': 1}

def test_freq_palavras_694():
    assert freq_palavras('   value ') == {'value': 1}
```

Logo depois filtramos essa grande quantidade de testes, retornando um conjunto de testes menor mas com a mesma quantidade de requisitos satisfeitos, como apresentado na Figura 6. Para melhor compreensão do filtro dos testes, outras informações são apresentadas. A primeira informação apresentada são os requisitos de testes. Em seguida, apresentamos os testes junto e os requisitos satisfeitos. Em verde são os que nenhum outro teste satisfez, enquanto que e em cinza escuro são os já satisfeitos por algum outro teste. Sendo assim, o primeiro teste cobriu 7 requisitos e os dois testes posteriores cobri-

ram 1 requisito novo cada um. No total, 9 de 10 requisitos foram satisfeitos somente na etapa do CrossHair.

Figura 6 – Testes Filtrados do CrossHair

```
10 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4), (1, 2, 5), (2, 4, 1), (2, 5, 1), (4, 1, 2), (4, 1, 3),
(5, 1, 2), (5, 1, 3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4) (1, 2, 5) (2, 4, 1) (2, 5, 1) (4, 1, 3) (5, 1, 2) -> 7/10
def test_freq_palavras():
    assert freq_palavras('another another') == {'another': 2}

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 5) (2, 5, 1) (5, 1, 2) (5, 1, 3) -> 1/10
def test_freq_palavras_387():
    assert freq_palavras('value another') == {'value': 1, 'another': 1}

Novos requisitos satisfeitos pelo teste: (0, 1, 3) -> 1/10
def test_freq_palavras_6():
    assert freq_palavras('') == {}

Total de requisitos satisfeitos: 9/10. Percentual de cobertura: 90.0%
```

A próxima etapa tem como foco a requisição ao ChatGPT para melhorar e possivelmente incrementar os testes. Essa etapa consiste na criação do prompt que será enviado como payload à requisição. Com o template do prompt, descrito no Anexo A, basta preenchê-lo com os parâmetros de entradas da nossa ferramenta: o enunciado, tema e solução do problema assim como os testes filtrados do CrossHair. Com a resposta do ChatGPT, descrito na Figura 7, validamos e apresentamos em tela os testes retornados. Em seguida, filtramos e mostramos o resultado, seguindo a mesma lógica dos testes filtrados do CrossHair.

Figura 7 – Testes do ChatGPT

```
def test_freq_palavras_1():
    assert freq_palavras("dinheiro é dinheiro e vice versa") == {"dinheiro": 2, "é": 1, "e": 1, "vice": 1, "versa": 1}

def test_freq_palavras_2():
    assert freq_palavras("python python python") == {"python": 3}

def test_freq_palavras_3():
    assert freq_palavras("teste teste teste teste") == {"teste": 4}

Testes ChatGpt Filtrados

10 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4), (1, 2, 5), (2, 4, 1), (2, 5, 1), (4, 1, 2), (4, 1, 3), (5, 1, 2),
(5, 1, 3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4) (1, 2, 5) (2, 4, 1) (2, 5, 1) (4, 1, 2) (5, 1, 2) (5, 1, 3) -> 8/10
def test_freq_palavras_1():
    assert freq_palavras("dinheiro é dinheiro e vice versa") == {"dinheiro": 2, "é": 1, "e": 1, "vice": 1, "versa": 1}

Total de requisitos satisfeitos: 8/10. Percentual de cobertura: 80.0%
```

Neste exemplo, o CrossHair retornou 3 testes, 2 deles redudantes. Por fim, mostramos o resultado final, combinando os esforços da etapa do CrossHair com a etapa do ChatGPT, representado na Figura 8. Três testes foram necessários para atingir 100% de cobertura, 2 provenientes do CrossHair e 1 do ChatGPT. Mostramos também algumas métricas, como o custo atrelado ao uso da API e o tempo de execução de cada etapa.

Figura 8 – Resultado Final

```
10 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4), (1, 2, 5), (2, 4, 1), (2, 5, 1), (4, 1, 2), (4, 1, 3), (5, 1, 2),
(5, 1, 3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4) (1, 2, 5) (2, 4, 1) (2, 5, 1) (4, 1, 2) (5, 1, 2) (5, 1, 3) -> 8/10
def test_freq_palavras_1():
    assert freq_palavras("dinheiro é dinheiro e vice versa") == {"dinheiro": 2, "é": 1, "e": 1, "vice": 1, "versa": 1}

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4) (1, 2, 5) (2, 4, 1) (2, 5, 1) (4, 1, 2) (4, 1, 3) (5, 1, 2) -> 1/10
def test_freq_palavras_2():
    assert freq_palavras("python python python") == {"python": 3}

Novos requisitos satisfeitos pelo teste: (0, 1, 3) -> 1/10
def test_freq_palavras_6():
    assert freq_palavras('') == {}

Total de requisitos satisfeitos: 10/10. Percentual de cobertura: 100.0%
Custo da Requisição à api do ChatGpt: $0,000434
Tempo de Execução do CrossHair: 103,04s
Tempo de Execução do ChatGpt para o critério de par arestas: 2,61s
```

Com isso concluímos o exemplo da geração de testes para um problema da Machine Teaching, mostrando o funcionamento principal da nossa ferramenta. Ele evidencia o potencial da combinação da execução concólica com a inteligência artificial como ferramenta de geração de testes.

4.4 TEMPO DE EXECUÇÃO E CUSTOS

Questões relevantes ao processo de geração de teste são o tempo de execução e os custos monetários associados. A demora em gerar os testes, assim como um custo monetário elevado, podem inviabilizar o uso da ferramenta. Dessa forma, alcançar um bom tempo de execução e reduzir os custos do uso da inteligência artificial foram focos de estudo neste projeto.

Encontrar os parâmetros de execução do CrossHair a fim de otimizar a cobertura e minimizar o tempo de execução demandou vários experimentos. Entre todos os parâmetros de timeout do CrossHair, optamos por seguir com o `per_condition_timeout` com valor igual a 100. Aliado ao path coverage, ele traz ótimos resultados, e demora entre 1 minuto e meio a 2 minutos e meio para gerar os testes de um problema.

Além do custo computacional de processamento do CrossHair temos o custo das requisições ao GPT. Esse custo varia proporcionalmente à quantidade de tokens enviados e recebidos da requisição. Em geral, o custo dos tokens de entrada, enviados no prompt, é menor que o custo dos tokens de saída, que corresponde à resposta do GPT. A fim de reduzir custos, é mais vantajoso enviar mais tokens no prompt da requisição, fornecendo o contexto e descrevendo minuciosamente o formato de saída, para reduzir a quantidade de tokens retornados.

Além disso, o modelo do GPT influencia enormemente no custo de geração de testes de cada problema. Durante os testes deste projeto duas versões foram utilizadas: `gpt-3.5-turbo` e `gpt-4-turbo-preview`. A 3.5 foi amplamente utilizada durante o de-

senvolvimento, enquanto que a versão 4 foi utilizada durante três dias. A escolha do modelo 3.5 ocorreu devido à diferença de preço entre os modelos. Ao visualizar a tabela de preços do GPT da OpenAi, contida na Figura 9, vemos que o preço da versão 4 é 20 vezes superior ao preço da versão 3.5. Fora isso, o ganho de conhecimento e a habilidade de resolver problemas mais complexos do modelo 4 não justificam os custos atrelados ao seu uso.

Figura 9 – Custo dos Modelos do GPT Utilizados

Model	Input	Output
gpt-4-0125-preview	$0.01 / 1K tokens	$0.03 / 1K tokens
gpt-4-1106-preview	$0.01 / 1K tokens	$0.03 / 1K tokens
gpt-4-1106-vision-preview	$0.01 / 1K tokens	$0.03 / 1K tokens

Model	Input	Output
gpt-3.5-turbo-0125	$0.0005 / 1K tokens	$0.0015 / 1K tokens
gpt-3.5-turbo-instruct	$0.0015 / 1K tokens	$0.0020 / 1K tokens

Fonte: https://openai.com/pricing#language-models

A Figura 5 ilustra bem a diferença de custo do uso dos modelos. Analisando os três dias em que o modelo 4 foi usado observamos a diferença de $1.21 dólares para os $0.02 dólares do modelo 3.5. Nesse comparativo, 64 requisições ao modelo 3.5 foram feitas, totalizando 35936 tokens. No modelo 4, 65 requisições foram realizadas, totalizando 65510 tokens. Pensando nisso, mesmo dobrando a quantidade de tokens do modelo 3.5, o custo ficaria apenas em $0.04 dólares, aproximadamente 30 vezes menos que o gasto com o modelo 4.

Figura 10 – Custo dos Modelos do GPT Utilizados

Fonte: https://platform.openai.com/usage

5 ANÁLISE DOS RESULTADOS

Neste capítulo analisamos a nossa ferramenta, responsável pela geração automática de testes de unidade. Inicialmente dizemos quais são os objetivos que desejamos alcançar, para no final analisarmos se os alcançamos ou não.

Então atingimos a parte principal do capítulo, com os testes realizados sobre a performance da ferramenta. A partir de um conjunto de problemas da plataforma de aprendizado, podemos comparar a cobertura entre os novos testes e os testes originais da plataforma.

No final, apresentamos as limitações da nossa ferramenta encontradas durante o processo de teste e o custo associado ao seu uso.

5.1 OBJETIVOS

Neste trabalho nós geramos testes de unidade com o uso de critérios de cobertura baseados em grafos. Para verificar se a geração é satisfatória, ou seja, se a quantidade de testes gerados cobrirá boa parte dos fluxos de execução e, consequentemente, identificará grande parte das soluções defeituosas inseridas pelos alunos, temos os seguintes objetivos:

1. Atingir um percentual de cobertura de testes maior ou igual à cobertura dos testes atuais da plataforma de aprendizado Machine Teaching.

2. Alcançar um tempo médio razoável para gerar automaticamente os testes unitários de qualquer problema da plataforma Machine Teaching.

3. Minimizar o custo do uso da inteligência artificial para a geração de testes.

4. Verificar que a combinação de execução concólica e inteligência artificial supera o uso individual de cada uma, em relação à cobertura de testes e identificação de soluções defeituosas inseridas pelos alunos.

Através destes questionamentos poderemos deduzir se a ferramenta está fornecendo uma cobertura abrangente de testes para os problemas da plataforma Além disso, saberemos o tempo gasto para ter o benefício da automação, verificando se a usabilidade da ferramenta é viável. Conseguiremos dizer também se o novo conjunto de testes consegue identificar com sucesso as soluções defeituosas dos alunos. E, por fim, veremos se o uso de técnicas de execução simbólica são relevantes para a geração de testes de unidade se comparado com uma inteligência artificial.

5.2 EXPERIMENTOS REALIZADOS

Para os experimentos selecionamos 41 problemas de uma turma de introdução à programação da plataforma Machine Teaching. Estes mesmos problemas foram usados no trabalho anterior de Albuquerque, Boéchat e Coutinho, que também nos forneceram os gabaritos dos problemas (ALBUQUERQUE; BOÉCHAT; COUTINHO, 2023). Os problemas usados abrangem temas distintos de introdução à computação, como laços, strings, tuplas, dicionários, listas, matrizes, entre outros. Essa variação é importante para validar se a nossa ferramenta consegue ser útil mesmo nos cenários em que o critério de cobertura baseado em grafo não é favorável.

Dos 41 problemas, consideramos apenas 36. Os outros 5 apresentaram algum impeditivo e precisaram ser desconsiderados da nossa avaliação. Dois problemas dependiam de soluções de problemas anteriores, e a nossa avaliação ficaria distorcida dada a presença de dois problemas iguais, o problema original e a sua continuação. Os outros três problemas não tinham gabarito disponível, e sem este gabarito não podemos seguir com a geração de testes.

O processo experimental consistiu em obter e formatar todos os dados para o uso em nossa ferramenta. Os dados dos problemas são encontrados no arquivo `problem.py` no repositório do GitHub, cujo link está no Anexo 2 . Para cada problema, precisamos de 4 informações. A primeira é o gabarito do professor, obtida através do material disponível no Google Drive, cujo link está no Anexo 2. As outras três, disponíveis na plataforma MT, são o enunciado do problema, o tema dele e os testes já existentes. Como essas informações são apresentadas em HTML na plataforma, criamos um script de *web scraping* para obter as informações.

Os experimentos foram executados em um computador desktop, com um processador AMD Ryzen 7 5700x com velocidade base de 3.4 GHz, 32 GB de RAM DDR4 com velocidade de 2400 MHz, no sistema operacional Windows 11. Todas as etapas, exceto a requisição ao ChatGPT na nuvem, foram influenciadas pela configuração do computador.

Executamos a nossa ferramenta três vezes para cada problema. Para cada um, medimos a quantidade de requisitos satisfeitos na etapa do CrossHair, na etapa do ChatGPT e combinando o resultado dos dois. Além disso, medimos o tempo de execução e o preço pago pelas requisições ao ChatGPT. Os resultados destes experimentos estão descritos nas seções 5.3 e 5.4. Os resultados experimentais completos estão disponíveis em uma planilha no Anexo D.

5.3 ANÁLISE DOS RESULTADOS

Os resultados de cobertura consolidados estão apresentados no Quadro 1. A primeira coluna indica o número do problema na plataforma. Realizamos os testes para cada um dos três critérios de cobertura baseados em grafo: nós, arestas, e par de arestas. A

cobertura dos novos testes indicada é a média aritmética das três execuções do problema (visto que a nossa ferramenta não é determinística, devido ao uso do ChatGPT). Para cada critério, comparamos os testes atuais, preparados manualmente por um professor na MT, e os testes novos, gerados pela nossa ferramenta. Dessa forma, nas colunas, a sigla "A" significa atual enquanto que a sigla "N" significa novo. As partes em negrito indicam casos em que nossa ferramenta obteve uma cobertura diferente dos testes originais. As setas indicam se a variação foi positiva ou negativa.

De acordo com os resultados apresentados no Quadro 1 concluímos que a nossa ferramenta apresentou um desempenho bem satisfatório. Em todos os critérios tivemos um número maior de problemas em que a cobertura aumentou do que diminuiu. No critério de nós tivemos 2 problemas com variação positiva e 1 problema com variação negativa. No critério de arestas tivemos 4 positivos e 1 negativo. O melhor resultado obtido foi com o critério de par de arestas, com 13 casos positivos e 2 negativos. Este resultado coincide com o esperado, dado que o critério de par de arestas é mais rigoroso que os demais. Combinando os resultados, ou seja, considerando os novos testes nos 13 casos positivos e considerando os testes originais da plataforma nos 2 casos negativos, temos aumento de cobertura de testes em 36% dos problemas.

Vale ressaltar que os testes atuais dos problemas da MT já passaram por várias iterações. É possível que a primeira versão de testes que o professor criou tenha tido uma cobertura menor ainda. No entanto, não temos o históricos dos testes do MT para medir isto.

No geral, quando a diferença de cobertura entre os testes originais e os novos testes não é muito grande, ela está relacionada com casos de valores de entrada "vazio". Por exemplo, listas vazias, valor zero, string vazia, em alguns dos problemas são testes válidos mas que não foram considerados na MT. São casos no qual o CrossHair costuma considerar ao gerar os testes e, portanto, a nossa ferramenta conseguiu cobrir. Alguns exemplos da diferença de cobertura estão disponíveis no Anexo B.

Temos alguns casos interessantes de variação positiva para analisar: os problemas 800, 811 e 828.

O problema 800 consiste em calcular o total de uma compra em um supermercado. O problema teve uma grande disparidade de cobertura no critério de par de arestas pois os testes do MT consideraram apenas o cenário em que todos os produtos comprados estavam no dicionário de valores. Dessa forma, apenas 5 dos 9 requisitos foram satisfeitos. O CrossHair considerou dois cenários novos: quando não existe nenhum item comprado e quando o item comprado não está no dicionário de valores. Com esses dois casos, 4 requisitos são satisfeitos. O GPT considerou o cenário no qual os produtos na lista de compra estão no dicionário de valores, satisfazendo os outros 5 requisitos. Ao uní-los, 9 dos requisitos foram satisfeitos, em detrimento dos 5 requisitos satisfeitos pelos testes existentes do MT.

Quadro 1 – Comparativo de Cobertura (%) Testes Atuais e Novos - Todos os Critério

Problema	Nós A.	Nós N.	Arestas A.	Arestas N.	Par Arestas A.	Par Arestas N.
734	**100**	**89** ↓	**100**	**87** ↓	**100**	**86** ↓
735	100	100	100	100	100	100
736	100	100	100	100	100	100
742	100	100	100	100	100	100
744	100	100	100	100	100	100
751	100	100	100	100	100	100
798	100	100	100	100	**80**	**100** ↑
800	100	100	**83**	**100** ↑	**44**	**100** ↑
802	100	100	100	100	100	100
804	100	100	100	100	100	100
806	100	100	100	100	100	100
807	100	100	100	100	100	100
808	100	100	100	100	100	100
809	100	100	100	100	100	100
811	**83**	**100** ↑	**80**	**100** ↑	**75**	**100** ↑
812	100	100	100	100	100	100
815	100	100	100	100	100	100
816	100	100	100	100	100	100
819	100	100	100	100	**89**	**100** ↑
820	100	100	100	100	**83**	**100** ↑
821	100	100	100	100	**80**	**100** ↑
822	100	100	100	100	**89**	**100** ↑
823	100	100	**80**	**100** ↑	**80**	**100** ↑
824	100	100	100	100	**90**	**100** ↑
827	100	100	100	100	100	100
828	**75**	**100** ↑	**75**	**100** ↑	**78**	**100** ↑
829	100	100	100	100	**80**	**100** ↑
831	100	100	100	100	**90**	**100** ↑
832	100	100	100	100	100	100
833	100	100	100	100	100	100
834	100	100	100	100	100	100
835	100	100	100	100	**67**	**44** ↓
836	100	100	100	100	**89**	**100** ↑
838	100	100	100	100	100	100
839	100	100	100	100	100	100
840	100	100	100	100	100	100

Os problemas 811 e 828 são problemas em que todos os critérios tiveram uma variação positiva. Neles, if's que a solução possui não foram explorados. Por exemplo o problema 828, que determina se um número é primo, não tem casos de teste para um número menor que 2 ou igual a 2, embora a solução contenha tratativas específicas para esses dois cenários.

Os dois problemas com variação negativa são interessantes de analisar. O problema

734, que verifica entre dois times de futebol qual está melhor classificado em um campeonato, teve uma variação negativa em todos os critérios. Isso ocorreu pois a nossa ferramenta não conseguiu gerar um caso de teste que entrasse em um `if` em específico. Dado que todos os experimentos deste problema isso aconteceu, parece uma limitação da nossa ferramenta. Talvez, se deixássemos o CrossHair executando durante mais tempo, ele encontrasse esse cenário.

O problema 835 consiste em uma limitação da nossa ferramenta. A entrada do problema consiste em uma matriz 6 x 10 de inteiros. Mesmo com o uso de typehint indicando que é uma lista de lista de inteiros, o CrossHair teve dificuldade em gerar ao menos um caso de teste que estivesse de acordo com a entrada. O GPT, por interpretar o contexto, conseguiu gerar testes com a entrada correta, mas em geral não produziu o resultado correto. Dessa forma, quando validamos os testes do GPT, acabamos descartando os testes dado que a saída fornecida não condiz com a saída obtida ao executar a solução do professor. Em um trabalho futuro vamos considerar apenas as entradas e obter as saídas executando a solução do professor.

Outro ponto que ainda não falamos é a relação entre o CrossHair e o ChatGPT. Analisamos se a combinação das duas técnicas supera o uso separado delas. Nos experimentos separamos a quantidade de testes criados em cada uma das técnicas assim como o número de requisitos satisfeitos. Então, combinamos o melhor das duas técnicas, melhorando o resultado final Essas informações estão dispostas no Quadro 2, por meio de 7 colunas, que contém o identificador do problema, o número de testes criados pelo CrossHair e o total de requisitos satisfeitos por ele, em seguida a mesma lógica mas para o ChatGPT, e por último a combinação dos dois.

Vale notar que consideramos apenas os resultados do critério de par de arestas. Também não incluímos os problemas em que o número de requisitos satisfeitos pelo CrossHair coincidiu com o número de requisitos satisfeitos pelo ChatGPT. Em negrito estão destacados os problemas que, em questão de cobertura, o uso das duas técnicas superou o uso individual. A partir do conjunto de 36 problemas, 3 apresentaram um aumento de cobertura com o uso de ambas as técnicas.

Em geral, tivemos uma cobertura maior do CrossHair em relação ao GPT. Este é um resultado esperado, visto que em nossa ferramenta o CrossHair é principal responsável por obter os casos de testes. O intuito do uso da inteligência artificial é de complementar os casos de testes obtidos, alterando as entradas para serem mais intuitivas. Além disso, configuramos o ChatGPT com um prompt mais restrito e um baixo parâmetro de criatividade, para que ele ficasse próximo dos casos de teste propostos pelo CrossHair.

Se usássemos somente o CrossHair teríamos uma cobertura menor e testes menos intuitivos. Por outro lado, a inteligência artificial sozinha provavelmente resultaria em uma menor cobertura de testes.

Quadro 2 – Requisitos Satisfeitos Critério de Par de Arestas: CrossHair e ChatGPT

Problema	#CrossHair	Requisitos	#GPT	Requisitos	#Total	Requisitos
734	4	6	3	5	4	6
798	3	9	1.7	8.7	3	**10**
804	8	18	5	16.3	8	18
806	3	4	2	3	3	4
811	3	4	2	3	3	4
819	2	9	1	6	2	9
820	4	12	1	9	4	12
821	2	5	1	4	2	5
822	2	9	1	8	2	9
823	2	5	1	4	2	5
824	2	10	0	0	2	10
828	4	8	2	6	4	8
831	3	10	1	8	3	10
833	1	2.3	1.7	4.7	2	**5**
835	0	0	0.3	1.3	0.3	**1.3**
836	3	9	1	5	3	9

5.4 TEMPO DE EXECUÇÃO E CUSTOS

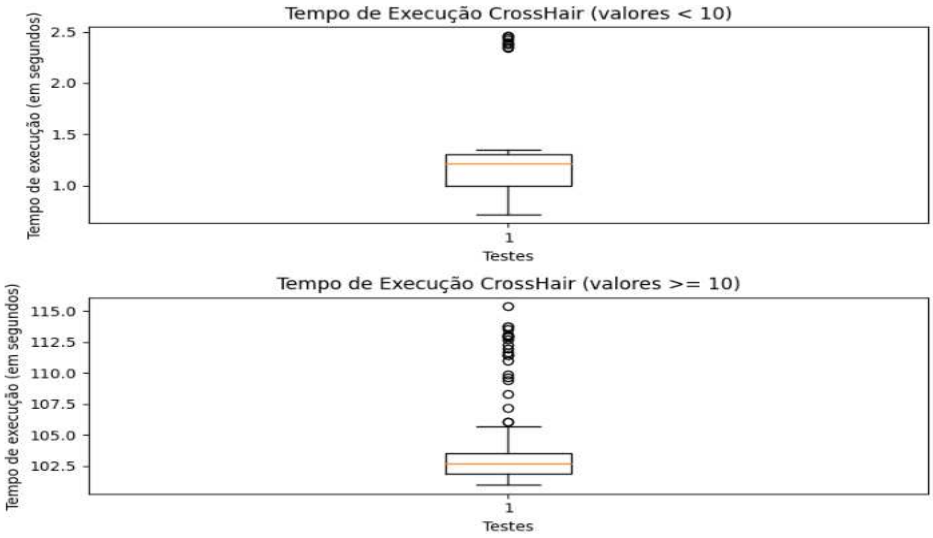
5.4.1 Tempo de Execução

Analisando o tempo de execução do CrossHair, verificamos que ou ele roda muito rápido, levando menos de 2.5 segundos, ou ele atinge o tempo máximo que definimos, entre 115 e 100 segundos. Definimos o tempo limite para exploração de um caminho de 100 segundos, mas no total o CrossHair pode levar um tempo a mais para rodar. O motivo para seguir um caso ou outro ainda não nos é muito claro. O CrossHair só vai parar de rodar se identificar que explorou tudo que tinha para explorar. Para isso, ele precisa saber que usou o tipo certo e que não há mais nenhum tipo para usar e que todos os caminhos viáveis foram percorridos.

Dada a discrepância do tempo de execução do CrossHair separamos em dois casos: tempo de execução menor que 10 segundos e tempo de execução maior ou igual a 10 segundos. O resultado é apresentado através do Box Plot na Figura 11. Temos 53 experimentos que o tempo é menor que 10 segundos, com uma média de 1.21 segundos, enquanto que a média para os outros 273 casos o tempo é de 1 minuto e 43 segundos. Vemos uma dispersão maior no cenário em que o CrossHair roda até seu limite. Levando em consideração os 324 experimentos, o tempo médio foi de 1 minuto e 27 segundos e o tempo total foi de 7 horas, 51 minutos e 26 segundos.

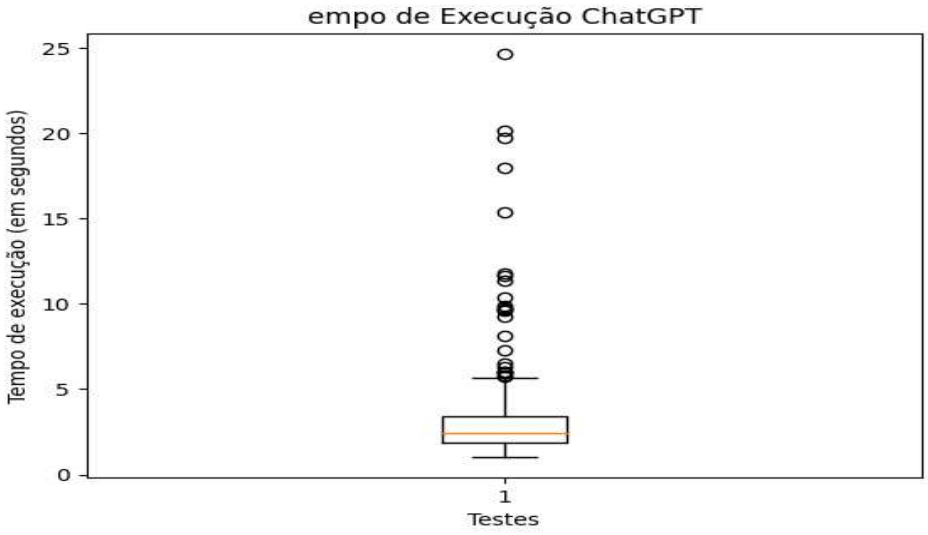
Analisando o tempo de execução do GPT, não temos cenários tão discrepantes se comparado ao CrossHair. O tempo é apresentado através da Figura 12, com uma média de 2.4 segundos para obter a resposta do ChatGPT, e um total de 17 minutos considerando

Figura 11 – Tempo de Execução CrossHair (em segundos)



todos os experimentos realizados.

Figura 12 – Tempo de Execução ChatGPT (em segundos)

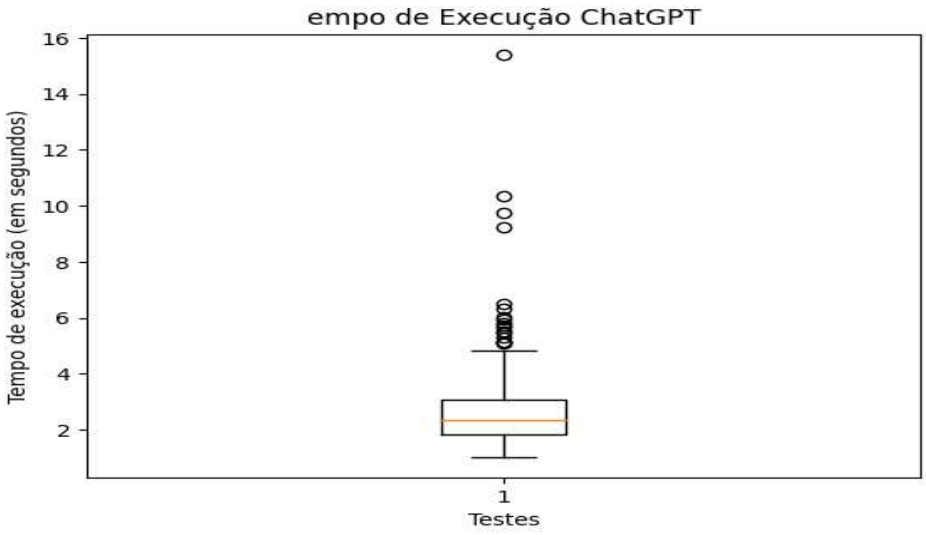


Ao analisarmos os outliers, percebemos que problemas com entradas maiores levaram mais tempo na resposta dado o trabalho de escrever essas entradas. Olhando para o problema 835, que calcula qual foi a melhor volta a partir do tempo de corredores, o tempo médio de resposta foi de 15.20 segundos. Acontece que a entrada é uma matriz 6x10, e o GPT precisa construir respostas passando matrizes grandes, o que leva mais tempo. Desconsiderando este problema e o 836, que também recebe matrizes de entrada, verificamos pela Figura 13 uma presença menor de outliers.

O experimento que demorou 15,4 segundos, maior outlier da Figura 13, teve um tempo médio de 4.37 segundos, indicando o cenário em que a latência de resposta do ChatGPT é maior por conta da grande demanda ao servidor. Para esse problema, o critério de nós

levou em média 9.03 segundos, enquanto que o critério de arestas levou, em média, 2.21 segundos e o critério de par de arestas 1.98 segundos. Quando executamos os experimentos para o critério de nós para esse problema o servidor do GPT poderia estar sobrecarregado, resultado em tempos maiores.

Figura 13 – Tempo ChatGPT Sem problemas 835 e 836 (em segundos)



Combinando os tempos das duas etapas saberemos o tempo total para realizar todos os experimentos e a média de tempo esperada para criar os testes de um problema. A média foi de 1 minuto e 45 segundos e o tempo para realizar todos os experimentos foi de 8 horas, 8 minutos e 25 segundos. Vale notar que o tempo mínimo para criação dos testes para um problema foi de 2 segundos e o tempo máximo foi de 2 minutos e 7 segundos, um limite razoável para a criação de testes dado que com intervenção manual provavelmente teríamos um esforço maior.

5.4.2 Custo do GPT

Além do custo computacional de rodar a ferramenta temos o custo monetário das requisições ao serviço do ChatGPT. Tal custo varia de acordo com o modelo utilizado e com a quantidade de tokens de entrada e saída. Em nossos experimentos usamos somente o modelo 3.5-turbo, e o custo para cada problema está disponível na planilha do Anexo D. A variação de custo está sujeita à variação entre os problemas e não entre os critérios. Portanto, esperamos para um mesmo problema, obter custos muito semelhantes entre os critérios.

Consolidando todos os experimentos, tivemos um custo total de aproximadamente \$0.16. É um custo bem baixo se consideramos a quantidade de experimento que realizamos. O cenário poderia ser bem mais caro se usássemos o modelo 4. No geral, o modelo 3.5-turbo satisfaz nossas necessidades, com boas respostas e baixo tempo de execução e custo.

5.5 LIMITAÇÕES ENCONTRADAS DURANTE O PROCESSO DE TESTE

Encontramos algumas limitações da nossa ferramenta durante o processo de avaliação. Essas limitações impactaram os testes, tanto pelo tempo de análise quanto pela intervenção manual requirida.

Nossa ferramenta considera cada linha de código como apenas um comando. No entanto, o Python permite mais de um comando por linha. Nesses cenários, é possível que a nossa ferramenta não crie corretamente os testes de unidade ou indique uma cobertura diferente da realidade. Nossa ferramenta também não é robusta com pós-processamento de código.

O CrossHair faz uso de typehints para escolher melhor os valores de entradas. No entanto, em alguns casos ele não interpretou corretamente os tipos, gerando erro na execução do comando. Nestes casos removemos parcialmente ou por completo o uso dos typehints. Sem a indicação dos tipos, o desempenho da geração de testes do CrossHair é prejudicada, influenciando na cobertura de testes.

Outra limitação encontrada no CrossHair é o uso da função `sort`. Tivemos que intervir manualmente nas soluções que usam a função `list.sort()`, alterando para `sorted(list)`. Acontece que o CrossHair transforma a lista em uma SymbolicList, uma lista de valores simbólicos, e essa lista não aceita o uso da função `sort`. No entanto, a função `sorted()` funciona normalmente. O mesmo acontece para a função `list.reverse()`, que precisa ser alterada para `reversed(list)`. No conjunto de 36 problemas, apenas uma solução fez uso dessas funções, e foi alterada para seguir com a geração dos testes.

Além disso, fizemos uso do gerador de grafo de controle de fluxo desenvolvido por Albuquerque, Boéchat e Coutinho. Esse gerador faz uma transformação no código original que impactou o processo de testes. A primeira transformação identificada foi a tradução de elif para if. Como o comando estava mapeado como elif na nossa ferramenta e apenas if no grafo de controle de fluxo, tivemos erro ao obter os nós de execução de acordo com a lista de comandos. A segunda transformação foi a remoção dos parêntesis de if's. Por exemplo, o seguinte comando `if (1 > 2):` foi traduzido para `if 1 > 2:`. Isso também impactou o mapeamento da lista de comandos para a lista de nós do grafo de controle de fluxo.

Voltando o olhar para a etapa do ChatGPT, também temos uma limitação. Em alguns cenários a inteligência artificial pode não compreender o contexto do problema, apresentando saídas fora do esperado. Todo teste fornecido pelo ChatGPT é validado executando a solução do professor e verificando se a saída esperada que o ChatGPT indicou coincide com o resultado da execução da solução. Se não coincidir nossa ferramenta descarta o teste. Isso ocorre pois solicitamos ao GPT o retorno esperado e o consideramos em nossa análise. O mais correto, e que traria maior cobertura para a nossa ferramenta, seria pedir apenas os valores de entradas e obter os valores de saídas a partir da execução

da solução do professor.

Observamos essa ocorrência em dois problemas, 824 e 835. O 834 recebe uma frase e o retorno esperado é a mesma frase mas com todas as consoantes em maiúsculo. Mesmo com o enunciado esclarecendo isso, o ChatGPT entendeu o contrário, e retornou as frases com todas as vogais maiúsculas. O problema 835 calcula a melhor volta feita dentre 6 corredores de kart, e o GPT retornou boas entradas. No entanto, as saídas não condiziam com as entradas. Se refizéssemos toda a ferramenta, não teríamos solicitado ao ChatGPT as saídas, somente as entradas, e teríamos calculado os valores de saída a partir do gabarito do problema.

6 CONCLUSÃO

Neste trabalho desenvolvemos uma ferramenta para criação de testes de unidade em Python com o uso de critérios baseados em grafos. Para medir a viabilidade do seu uso definimos quatro objetivos a serem alcançados pela nossa ferramenta: atingir um percentual de cobertura de testes maior ou igual à cobertura dos testes da MT, verificar que a combinação de execução concólica e inteligência artificial supera o uso individual de cada uma, alcançar um tempo médio razoável de execução e minimizar o custo do uso da inteligência artificial.

Para medir o desempenho, rodamos a nossa ferramenta em um conjunto de problemas da plataforma Machine Teaching. Comparamos a quantidade de requisitos satisfeitos dos testes criados em relação a quantidade de requisitos satisfeitos pelos testes atuais da MT. Nos três critérios: nós, arestas e par de arestas, obtivemos uma cobertura maior que os testes originais da plataforma. Em especial, o critério de par de arestas apresentou maior diferença, atingindo um percentual de cobertura de testes 36% maior em relação aos testes da MT.

Atingimos esse feito através do uso de duas técnicas: execução concólica e LLM. Verificamos que a combinação das duas potencializou a cobertura de testes e que o uso de individual de cada uma provavelmente não alcançaria os mesmos resultados que a nossa ferramenta.

Por último, ao analisar os dados de todos os experimentos realizados, concluímos que a nossa ferramenta possui um baixo custo. No pior cenário o tempo de execução levou 2 minutos e 7 segundos, um tempo razoável para criação dos testes. O custo da inteligência artificial também se mostrou baixo, custando $0.16 dólares considerando todos os experimentos realizados. O valor entregue pela nossa ferramenta e o baixo custo mostram a viabilidade do seu uso em plataformas de aprendizado e possivelmente em outras áreas.

6.1 TRABALHOS FUTUROS

Como próximos passos da nossa ferramenta vemos alguns caminhos que podem aumentar ainda mais a cobertura de testes. Além disso, utilizamos a nossa ferramenta no contexto da plataforma de aprendizado Machine Teaching. Outros contextos podem ser testados para avaliação do desempenho e possíveis melhorias.

Como foco, temos:

- Ao receber a resposta do ChatGPT considerar somente os valores de entradas e obter os valores de saída através da solução do professor. Assim, sempre poderemos

aproveitar os testes do ChatGPT, até quando ele chuta uma resposta que não condiz com o problema.

- Realizar os mesmos experimentos mas alterando os parâmetros do CrossHair e do ChatGPT a fim de alcançar a melhor configuração possível. Neste trabalho realizamos alguns testes, mas em um conjunto pequeno de problemas.
- Avaliar com o principal contribuidor do CrossHair o motivo de alguns programas com type hints não funcionarem com o CrossHair.
- Realizar os mesmos experimentos considerando as soluções dos alunos em vez da quantidade de requisitos satisfeitos. Dessa forma, teremos a quantidade de soluções defeituosas que os nossos testes detectaram.
- Analisar as características dos problemas que apresentaram divergência na cobertura.
- Testar a ferramenta em programas mais complexos e analisar os resultados obtidos.
- Disponibilizar uma ferramenta gráfica para execução da ferramenta e integrar à Machine Teaching

REFERÊNCIAS

ALBUQUERQUE, A. M. d.; BOÉCHAT, F. A.; COUTINHO, L. S. Modelagem de testes de software: uma análise dos resultados de testes em exercícios de programação. Universidade Federal do Rio de Janeiro, 2023.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing Second Edition**. Cambridge, United Kingdom; New York, NY, USA: Library of Congress Cataloguing in Publication Data, 2017.

BARR, E. T. et al. The oracle problem in software testing: A survey. **IEEE transactions on software engineering**, IEEE, v. 41, n. 5, p. 507–525, 2014.

CADAR, C. et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: **OSDI**. [S.l.: s.n.], 2008. v. 8, p. 209–224.

CADAR, C.; SEN, K. Symbolic execution for software testing: three decades later. **Communications of the ACM**, ACM New York, NY, USA, v. 56, n. 2, p. 82–90, 2013.

CHENG, E. **Binary Analysis and Symbolic Execution with angr**. Tese (Doutorado) — PhD thesis, The MITRE Corporation, 2016.

CLARKE, L. A. A program testing system. In: **Proceedings of the 1976 annual conference**. [S.l.: s.n.], 1976. p. 488–491.

JAFFAR, J.; LASSEZ, J.-L. Constraint logic programming. In: **Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages**. [S.l.: s.n.], 1987. p. 111–119.

KING, J. C. Symbolic execution and program testing. **Communications of the ACM**, ACM New York, NY, USA, v. 19, n. 7, p. 385–394, 1976.

KIRMANI, A. R. Artificial intelligence-enabled science poetry. **ACS Energy Letters**, ACS Publications, v. 8, n. 1, p. 574–576, 2022.

LIU, X. et al. GPT understands, too. **AI Open**, Elsevier, 2023.

LUCKOW, K. **A curated list of awesome symbolic execution resources including essential research papers, lectures, videos, and tools**. 2023. Disponível em: https://github.com/ksluckow/awesome-symbolic-execution.

MCGRAW, G. Silver bullet talks with Bart Miller. **IEEE Security & Privacy**, IEEE, v. 12, n. 5, p. 6–8, 2014.

MONTANDON, L. Exhaustive symbolic execution engine for verifying python programs. **EPFL Switzerland**, EPFL, 2022.

MORAES, L. O. et al. Machine Teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação. In: SBC. **Anais do II Simpósio Brasileiro de Educação em Computação**. [S.l.], 2022. p. 213–223.

MOURA, L. D.; BJØRNER, N. Z3: an efficient smt solver. In: **Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. Berlin, Heidelberg: Springer-Verlag, 2008. (TACAS'08/ETAPS'08), p. 337–340. ISBN 3540787992.

OEHLERT, P. Violating assumptions with fuzzing. **IEEE Security Privacy**, v. 3, n. 2, p. 58–62, 2005.

Phillip Schanely. **CrossHair, An analysis tool for Python that blurs the line between testing and type systems**. 2024. https://github.com/pschanely/CrossHair/. Online; accessed 09 March 2024.

SEN, K. Concolic testing. In: **Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering**. [S.l.: s.n.], 2007. p. 571–572.

ANEXOS

ANEXO A – PROMPT DA REQUISIÇÃO AO CHATGPT

```
Vou te passar testes em pytest para um problema de programação em python
sobre o tema (Estrutura de repetição iteradora: for). Em seguida te
passarei o contexto do problema e a solução. Quero que você use os meus
testes, aqueles que forneci, mas altere os parâmetros para valores que
fazem sentido no problema dado, apenas valores que aparecem no problema real.
É muito importante que cada teste tenha somente um assert, se tiver mais
de um assert é necessario botar em outro teste com nome do teste diferente.

Meus Testes:
def test_freq_palavras_20():
    assert freq_palavras('  ') == {}
def test_freq_palavras_425():
    assert freq_palavras('   test') == {'test': 1}
def test_freq_palavras():
    assert freq_palavras('another another') == {'another': 2}
def test_freq_palavras_46():
    assert freq_palavras('another') == {'another': 1}

Contexto do Problema:
Construa uma função chamada **freq_palavras(frases)** que receba uma string e
retorne um dicionário onde cada palavra dessa string seja uma chave e tenha
como valor o número de vezes que a palavra aparece. Por exemplo:

- freq_palavras("dinheiro é dinheiro e vice versa")

Retorna o dicionário: { "dinheiro":2, "é": 1, "e": 1, "vice": 1, "versa":1}

Solução do problema em python:
def freq_palavras(frase: str) -> dict:
    dic = {}
    lista = frase.split()
    for palavra in lista:
        if palavra in dic:
            dic[palavra] += 1
        else:
            dic[palavra] = 1
    return dic

É MUITO IMPORTANTE que a quantidade de testes se mantenha. Só é válido adicionar
mais testes se eles percorrem caminhos de execução diferentes!!
Por fim, ponha os testes neste formato:
def test_1():
   assert...

def test_2():
   assert...

def test_3():
   assert...
```

ANEXO B – EXEMPLOS ONDE A NOSSA FERRAMENTA OBTEVE COBERTURA MAIOR QUE A MACHINE TEACHING

Além do exemplo principal deste trabalho, de contar a frequência das palavras em uma frase, temos outros exemplos interessantes. Analisaremos dois exemplos onde a cobertura da nossa ferramenta superou a cobertura dos testes da Machine Teaching considerando o critério de par de arestas.

B.0.1 Exemplo 1

Problema 800: Recebe uma lista de produtos comprados e um dicionário com o preço de cada produto e deve retornar o preço total da compra.

Por mais que a Machine Teaching tenha 10 testes, 9 passam pelo mesmo fluxo de execução. Portanto, após o filtro, apenas um teste é considerado, satisfazendo 5 dos 9 requisitos. O teste em questão percorre o fluxo mais comum, no qual a lista de produtos e os preços estão preenchidos, como demonstrado na Figura 14.

Figura 14 – Testes Machine Teaching Problema 800

```
9 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4),
(1, 2, 1), (2, 4, 1), (2, 1, 2), (2, 1, 3), (4, 1, 2), (4, 1,
3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4)
(2, 4, 1) (4, 1, 2) (4, 1, 3) -> 5/9

def test_case_1():
    assert total(['Chocolate pó', 'Azeitonas', 'Lentilha',
'Molho tomate', 'Creme leite'], {'Molho tomate': 8.31,
'Maizena': 8.23, 'Chocolate pó': 6.81, 'Azeitonas': 9.51,
'Lentilha': 4.11, 'Gelatina pó': 9.11, 'Orégano': 1.47, 'Creme
leite': 3.34, 'Palmito': 8.22}) == 32.08

Total de requisitos satisfeitos: 5/9.
```

Considerando os testes da nossa ferramenta, o CrossHair obteve dois fluxos de execução que a Machine Teaching não considerou. Um no qual a lista de compras possui produtos que não estão mapeados no dicionário de preços, e o segundo onde a lista de produtos comprados está vazia. Na figura 15, o primeiro teste corresponde ao primeiro cenário, e o segundo teste corresponde ao segundo cenário.

O ChatGPT criou testes que passam pelo mesmo fluxo de execução que a MT, como demonstrado na Figura 16.

Figura 15 – Testes CrossHair Problema 800

```
9 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4),
(1, 2, 1), (2, 4, 1), (2, 1, 2), (2, 1, 3), (4, 1, 2), (4, 1,
3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 1)
(2, 1, 2) (2, 1, 3) -> 4/9
def test_total_4():
    assert total([0, 0], {}) == 0

Novos requisitos satisfeitos pelo teste: (0, 1, 3) -> 1/9
def test_total_2():
    assert total([], {}) == 0

Total de requisitos satisfeitos: 5/9.
```

Figura 16 – Testes ChatGPT Problema 800

```
9 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4),
(1, 2, 1), (2, 4, 1), (2, 1, 2), (2, 1, 3), (4, 1, 2), (4, 1,
3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4)
(2, 4, 1) (4, 1, 2) (4, 1, 3) -> 5/9
def test_total_1():
    assert total(['arroz', 'cafe'], {'arroz': 10.90, 'cafe':
6.98}) == 17.88

Total de requisitos satisfeitos: 5/9.
```

Ao combinar o resultado das duas ferramentas, satisfazemos todos os requisitos, obtendo uma cobertura superior à MT, conforme a Figura 17. O aumento de cobertura é fruto do CrossHair conseguir explorar esses caminhos alternativos dos programas.

B.0.2 Exemplo 2

Problema 828: Recebe um número inteiro e determina se ele é primo.

Para esse problema os testes da Machine Teaching consideraram apenas dois fluxos de execução, como demonstrado na Figura 18. Os testes validam um número que é primo e outro número que não é, ambos maiores que 2. No entanto, no código temos validações para valores menores ou iguais a 2. Portanto, a cobertura ficou abaixo do esperado, com apenas 6 requisitos satisfeitos.

Figura 17 – Testes Finais Problema 800

```
9 Requisitos a satisfazer:  [(0, 1, 2), (0, 1, 3), (1, 2, 4), (1, 2, 1), (2, 4,
1), (2, 1, 2), (2, 1, 3), (4, 1, 2), (4, 1, 3)]

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 4) (2, 4, 1) (4, 1, 2)
(4, 1, 3) -> 5/9
def test_total_1():
    assert total(['arroz', 'cafe'], {'arroz': 10.90, 'cafe': 6.98}) == 17.88

Novos requisitos satisfeitos pelo teste: (0, 1, 2) (1, 2, 1) (2, 1, 2) (2, 1, 3)
-> 3/9
def test_total_4():
    assert total([0, 0], {}) == 0

Novos requisitos satisfeitos pelo teste: (0, 1, 3) -> 1/9
def test_total_2():
    assert total([], {}) == 0

Total de requisitos satisfeitos: 9/9.
```

Figura 18 – Testes Machine Teaching Problema 828

```
9 Requisitos a satisfazer:  [(0, 1), (0, 2, 3), (0, 2, 4), (2, 4, 5), (2, 4, 6),
(4, 5, 7), (4, 5, 4), (5, 4, 5), (5, 4, 6)]

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 7) (4, 5, 4)
(5, 4, 5) -> 5/9
def test_case_4():
    assert primo(121) == False

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 4) (5, 4, 5)
(5, 4, 6) -> 1/9
def test_case_6():
    assert primo(263) == True

Total de requisitos satisfeitos: 6/9.
```

A nossa ferramenta considerou os casos com número menor ou igual a 2. O primeiro caso é quando o número é igual a 2, retornando "true". O segundo caso é quando o número é menor que 2, retornando "false". Como apresentado na Figura 19, dos 9 requisitos 8 foram satisfeitos apenas na etapa do CrossHair. Como requisito faltante (2,4,6) é insatisfatível, o CrossHair conseguiu satisfazer todos os requisitos viáveis.

A cobertura de testes da I.A é parecida com a MT, cobrindo os dois fluxos no qual o número é maior que 2. Ao combinar os resultados das duas técnicas da nossa ferramenta, alcançamos a cobertura de todos os requisitos viáveis, como demonstrado na Figura 20.

Figura 19 – Testes CrossHair Problema 828

```
9 Requisitos a satisfazer:  [(0, 1), (0, 2, 3), (0, 2, 4), (2, 4, 5), (2, 4, 6),
(4, 5, 7), (4, 5, 4), (5, 4, 5), (5, 4, 6)]

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 7) (4, 5, 4)
(5, 4, 5) -> 5/9
def test_primo():
    assert primo(49) == False

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 4) (5, 4, 5)
(5, 4, 6) -> 1/9
def test_primo_5():
    assert primo(5) == True

Novos requisitos satisfeitos pelo teste: (0, 2, 3) -> 1/9
def test_primo_2():
    assert primo(2) == True

Novos requisitos satisfeitos pelo teste: (0, 1) -> 1/9
def test_primo_4():
    assert primo(0) == False

Total de requisitos satisfeitos: 8/9.
```

Figura 20 – Testes Finais Problema 828

```
9 Requisitos a satisfazer:  [(0, 1), (0, 2, 3), (0, 2, 4), (2, 4, 5), (2, 4, 6),
(4, 5, 7), (4, 5, 4), (5, 4, 5), (5, 4, 6)]

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 4) (5, 4, 5)
(5, 4, 6) -> 5/9
def test_primo_7():
    assert primo(7) == True

Novos requisitos satisfeitos pelo teste: (0, 2, 4) (2, 4, 5) (4, 5, 7) (4, 5, 4)
(5, 4, 5) -> 1/9
def test_primo_9():
    assert primo(9) == False

Novos requisitos satisfeitos pelo teste: (0, 2, 3) -> 1/9
def test_primo_2():
    assert primo(2) == True

Novos requisitos satisfeitos pelo teste: (0, 1) -> 1/9
def test_primo_4():
    assert primo(0) == False

Total de requisitos satisfeitos: 8/9.
```

ANEXO C – EXEMPLOS DE SUBSTITUIÇÃO DAS ENTRADAS PELO LLM

Um ponto importante para plataformas de aprendizado são as entradas e as saídas dos casos de teste. É essencial que tais valores façam sentido no contexto do problema e que estejam próximos da realidade para melhor compreensão dos alunos. Sendo assim, vejamos alguns exemplos onde a nossa ferramenta fez uso de uma LLM para fornecer melhores valores de entrada.

C.0.1 Exemplo 1

Problema 742: Recebe uma string, uma posição da string e um caractere e retorna a mesma string mas com o caractere trocado na posição indicada.

Ao analisarmos os testes gerados pelo CrossHair, indicados pela Figura 21, verificamos que as entradas criadas não são muito intuitivas. Por outro lado, as entradas do ChatGPT já condizem mais com a realidade do problema, como apresentado na mesma figura.

Figura 21 – Testes CrossHair e ChatGPT Problema 742

```
Testes CrossHair

def test_substitui():
    assert substitui('\x00', '', 0) == ''

def test_substitui_2():
    assert substitui('', '', 0) == 'i inválido'

Testes ChatGpt

def test_substitui():
    assert substitui('banana', 'x', 2) == 'baxana'

def test_substitui_2():
    assert substitui('hello', '!', 4) == 'hell!'
```

Combinando os resultados e removendo os testes redundantes, precisamos somente de um dois testes para satisfazer todos os requisitos. Dessa forma, o primeiro teste selecionado foi produzido pelo LLM, devido às entradas contextualizadas, e o segundo foi criado pelo CrossHair visto que o LLM não criou testes que percorrem esse fluxo, como demonstrado na Figura 22

Figura 22 – Testes Finais Problema 742

```
2 Requisitos a satisfazer:  [(0, 1), (0, 2)]

Novos requisitos satisfeitos pelo teste: (0, 2) -> 1/2
def test_substitui():
    assert substitui('banana', 'x', 2) == 'baxana'

Novos requisitos satisfeitos pelo teste: (0, 1) -> 1/2
def test_substitui_2():
    assert substitui('', '', 0) == 'i inválido'

Total de requisitos satisfeitos: 2/2.
```

C.0.2 Exemplo 2

Problema 744: Consiste em escrever uma função chamada hashtag que receba uma string e a retorne com o caractere '#' no início, no meio e no final dela.

Assim como o problema anterior, os testes gerados pelo CrossHair não possuem entradas muito intuitivas. A string em si não impacta na execução do programa, visto que basta adicionar '#' no início, meio e fim, mas queremos entradas contextualizadas.

Comparando os testes gerados pelo CrossHair e os testes da LLM como na Figura 23 observamos entradas simples, mas melhores.

Figura 23 – Testes CrossHair e ChatGPT Problema 744

```
Testes CrossHair

def test_hashtag():
    assert hashtag('') == '###'

def test_hashtag_2():
    assert hashtag('\x00') == '##\x00#'

Testes ChatGpt

def test_hashtag():
    assert hashtag('abcd') == '#ab#cd#'

def test_hashtag_2():
    assert hashtag('abcde') == '#ab#cde#'
```

Combinando os resultados e removendo os testes redundantes, precisamos somente de um teste para satisfazer todos os requisitos. O único teste selecionado é proveniente da LLM, dada suas entradas, como demonstrado na Figura 24.

Figura 24 – Testes Finais Problema 744

ANEXO D – LINKS PARA MATERIAL DO PROJETO

1. Todos os problemas que foram analisados estão formatados e armazenados em nossa ferramenta através do seguinte link: https://github.com/thierryxdp/TCC/blob/master/problems.py
2. Os resultados dos experimentos estão disponíveis na seguinte planilha: https://docs.google.com/spreadsheets/d/131K-ULCJkdQuyEJmzqMtIvR702ajj-WshCi8NaCW98Q/edit?usp=sharing
3. Gabaritos: os interessados nos gabaritos podem entrar em contato com o autor através do e-mail thierryxdp@dcc.ufrj.br.