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

ARTHUR MOREIRA DE ALBUQUERQUE
FRANÇOIS ALEXIS BOÉCHAT
LUCAS SENOS COUTINHO

MODELAGEM DE TESTES DE SOFTWARE

Uma Análise dos Resultados de Testes em Exercícios de Programação

RIO DE JANEIRO
2023

ARTHUR MOREIRA DE ALBUQUERQUE
FRANÇOIS ALEXIS BOÉCHAT
LUCAS SENOS COUTINHO

MODELAGEM DE TESTES DE SOFTWARE

Uma Análise dos Resultados de Testes em Exercícios de Programação

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: Prof. Anamaria Martins Moreira

RIO DE JANEIRO
2023

A345m

Albuquerque, Arthur Moreira de

Modelagem de testes de software: uma análise dos resultados de testes em exercícios de programação / Arthur Moreira de Albuquerque, François Alexis Boéchat e Lucas Senos Coutinho. – 2023.

56 f.

Orientadora: Anamaria Martins Moreira.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. Teste de software. 2. Critério de cobertura. 3. Plataforma de ensino online. I. Boéchat, François Alexis. II. Coutinho, Lucas Senos. III. Moreira, Anamaria Martins (Orient). IV. Universidade Federal do Rio de Janeiro, Instituto de Computação. V. Título.

ARTHUR MOREIRA DE ALBUQUERQUE
LUCAS SENOS COUTINHO
FRANÇOIS ALEXIS BOÉCHAT

MODELAGEM DE TESTES DE SOFTWARE
Uma Análise dos Resultados de Testes em Exercícios de Programação

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 20 de julho de 2023

BANCA EXAMINADORA:

Anamaria Martins Moreira
Doutora (UFRJ)

Carla Amor Divino Moreira Delgado
Doutora (UFRJ)

João Antonio Recio Paixão
Doutor (UFRJ)

Dedicatória: Dedicamos este trabalho para nossas famílias, por sempre demonstrarem determinação, integridade, resiliência e acima de tudo, amor.

AGRADECIMENTOS

Gostaríamos de agradecer à nossa orientadora, Anamaria Martins Moreira. Suas sugestões e correções mantiveram o trabalho no caminho certo. Além disso, sua disponibilidade durante o desenvolvimento do trabalho foi essencial para sua conclusão. Também agradecemos à equipe responsável pela plataforma Machine Teaching, que disponibilizou todas as informações e dados necessários para fazer a análise estatística. Um agradecimento especial a Maria Luíza Resende Damasceno, namorada de Lucas Senos Coutinho, cujo suporte emocional e apoio permitiu a conclusão desse trabalho. Além disso, somos gratos pelo professor João Antônio Recio da Paixão, que nos ajudou a encontrar o equilíbrio necessário para superar os desafios da jornada acadêmica. Por último, expressamos nossa gratidão por nossas famílias, cujo apoio incondicional viabilizou nosso progresso.

RESUMO

Todo período, alunos da UFRJ resolvem vários exercícios de programação que têm sua correção avaliada com testes na plataforma online Machine Teaching. Neste trabalho, testes foram elaborados usando técnicas sistemáticas de projeto de software. Esses testes foram aplicados nas soluções dos alunos para estudar os resultados de cada técnica. Para fazer isso, dados são extraídos ao se aplicar os testes, permitindo a comparação de cada técnica de modelagem de testes. A análise dos resultados avalia qual técnica de projeto de testes possui maior capacidade de identificação de defeitos inéditos nas soluções enviadas, assim como os erros de compilação mais frequentes e o custo de tempo para modelar cada tipo de teste. Como contribuição final, todo o arcabouço desse trabalho está disponibilizado para que novos experimentos possam ser feitos com base nele.

Palavras-chave: teste de software; critério de cobertura; plataforma de ensino online.

ABSTRACT

Every semester, UFRJ students solve various programming exercises that have their correctness evaluated using tests on the online platform Machine Teaching. In this research, new tests were developed using systematic software design techniques. These new tests were applied to the students solutions to study the results of each technique. To achieve this, data is gathered from the aplication of the new tests, allowing for the comparison of each technique. The analysis of the results measures which test design technique has the best capacity to identify new defects in the submitted solutions, as well as the most frequent errors and the time cost to design each type of test. As a final contribution, the documentation, data and source code of this work is available for future experiments to be conducted based on it.

Keywords: software testing; coverage criteria; online learning platform.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTO

Recentemente ocorreu uma proliferação de novas ferramentas que apresentam a possibilidade de melhorar o processo de aprendizagem. Entre essas ferramentas estão as plataformas de ensino online, que já são utilizadas para educação. Como alunos que utilizam plataformas de ensino apresentam melhor desempenho em relação a estudantes que não utilizam (MITROVIC; OHLSSON, 1999; BAKER, 2016), vale a pena estudar o efeito dessas plataformas no processo pedagógico. Desse modo, este trabalho utiliza dados reais retirados de uma dessas plataformas para alcançar melhor entendimento sobre aplicação prática de testes de software (MORAES; PEDREIRA, 2020).

Testes de software são procedimentos projetados para tentar identificar defeitos em programas. Ao facilitar a correção de defeitos por desenvolvedores, testes aumentam as chances do programa funcionar corretamente e de acordo com as expectativas de seus usuários. Testes indicam imediatamente em quais cenários um programa passou ou falhou. Essa informação ajuda o estudante a encontrar defeitos no código e pode ser utilizada no processo de ensino-aprendizagem. Para fazer isso, cada caso de teste é construído com uma saída esperada, que é predefinida. Ao executar a parte do código que está sendo verificada, o teste compara o resultado obtido do programa com a saída esperada e assim verifica se o programa atende as especificações do exercício.

Testes de software automatizam tarefas repetitivas que exigem tempo de docentes, como a correção de exercícios. Correção de exercícios apresentam maior custo de tempo em disciplinas com muitos exercícios e estudantes. Esse problema pode ser resolvido com testes automáticos para a avaliar exercícios feitos no decorrer da disciplina. Testes verificam exercícios feitos por alunos em menos tempo e checam se o exercício está de acordo com os critérios definidos pelo próprio professor. Desse modo, testes podem entregar feedback imediato para os alunos, facilitando assim o processo de ensino em disciplinas de programação.

No aprendizado de algumas disciplinas de programação da UFRJ, os alunos ganham acesso à plataforma Machine Teaching (MORAES,). Nessa plataforma cada estudante possui uma conta que recebe tarefas de programação elaboradas pelo professor. Como resposta, os alunos enviam códigos que terão sua funcionalidade verificada por meio de testes de software. Essa avaliação é essencial para checar se os alunos conseguiram compreender e aplicar os conceitos de programação ensinados na disciplina. O aluno pode rodar os testes na sua tentativa de solução e assim confirmar se ele atendeu todos os critérios esperados pelo professor. Desse modo, os testes podem oferecer uma avaliação instantânea para o estudante.

1.2 MOTIVAÇÃO

Testes de software identificam defeitos e reduzem custos de desenvolvimento (IBM,). Incorporar testes em disciplinas de programação agiliza a entrega de feedback para os alunos. O custo de tempo para modelar e implementar testes é um investimento, já que quaisquer padrões que forem identificados entre os defeitos podem ser reutilizados futuramente para encontrar defeitos semelhantes (EVERETT; MCLEOD, 2007). Testes automatizam a correção de exercícios e assim economizam tempo do instrutor. Para os alunos, os testes facilitam a identificação e o conserto de defeitos no código. Isto direciona os estudantes a desenvolverem programas que contém menos defeitos e que atendem todos os requisitos.

Na plataforma Machine Teaching, o instrutor tem bastante controle sobre a avaliação dos estudantes. Ele pode adicionar ou remover exercícios, assim como definir os testes que os alunos precisam passar em cada exercício. Esse controle na avaliação afeta a experiência dos alunos e a qualidade de ensino. Desse modo, quaisquer aperfeiçoamentos na avaliação ajudariam mais de 500 alunos que estão inscritos na plataforma Machine Teaching a cada semestre letivo. O feedback instantâneo descrevendo quais testes falharam são uma forma de apoiar o aluno no processo de aprendizado. Isso lhes dá a oportunidade de identificar critérios que não foram atendidos pela sua tentativa de solução e fazer os ajustes necessários para resolver o exercício corretamente.

Anteriormente, foram utilizados testes que ocasionalmente deixavam de cobrir cenários importantes, aceitando soluções com problemas como se fossem corretas. A aplicação de critérios rigorosos no projeto dos testes pode melhorar o desempenho e custo-benefício dos mesmos. Esse projeto almeja encontrar um bom compromisso entre maximizar a identificação de defeitos e elaborar um número moderado de testes (AMMANN; OFFUTT, 2016, p. 109). O refinamento da técnica de projeto de testes vai utilizar conceitos como requisitos de teste e critério de cobertura. Requisito de teste refere-se às condições ou características que algum caso de teste precisa cumprir. O critério de cobertura é um conjunto de regras que impõem requisitos que devem ser satisfeitos por um determinado conjunto de testes (AMMANN; OFFUTT, 2016, p. 57). Todos os novos testes foram modelados de acordo com 3 critérios de cobertura: particionamento de entrada, grafo e mutação. O uso desses critérios reduz o custo de extração de dados obtidos pelos testes (AMMANN; OFFUTT, 2016, p. 56), já que os testes são modelados para identificar erros de forma mais direcionada.

1.3 OBJETIVO

A proposta desse trabalho consiste em avaliar a diferença de desempenho ao se aplicar critérios rigorosos no projeto de testes em ferramentas pedagógicas como o Machine Teaching. A avaliação é feita com uma análise estatística onde os testes são separados por

critério de cobertura. Essa análise pode indicar onde existe espaço para aprimoramento e quando são necessários testes adicionais. Ao comparar cada critério será possível estimar qual critério deve ser priorizado em projetos futuros na modelagem dos testes, assim aperfeiçoando a modelagem de testes para melhor atender às necessidades de clientes em projetos futuros.

Para alcançar esse objetivo, será preciso:

- Definir as métricas que serão utilizadas na análise.
- Obter exercícios e soluções de alunos.
- Aplicar testes nas soluções.
- Extrair dados.
- Fazer a análise dos dados.

Os exercícios e soluções foram obtidos com o apoio da equipe de desenvolvimento da plataforma Machine Teaching. Esses dados não contém informações pessoais que poderiam identificam os autores de cada tentativa de solução (que neste trabalho chamaremos de solução, para simplificar, mesmo que seja apenas uma tentativa de solução), preservando o anonimato dos alunos. Todos os dados estão no arquivo pickles.zip, que contém todos os dados em arquivos no formato Pickle. O arquivo problems.pkl contém todos os enunciados dos problemas inseridos na plataforma Machine Teaching e o arquivo solutions.pkl contém todas as soluções enviadas por alunos por meio da plataforma. Esses dados estão disponíveis no Google Drive [1]. Depois de executar os testes projetados com base nos critérios de cobertura foco da pesquisa, extraímos e armazenamos os dados em um banco de dados, contendo os resultados de cada teste por problema e tentativa de solução por critério. Finalmente, utilizamos consultas SQL para estudar o desempenho e custo-benefício de cada critério. A partir disso, as conclusões da pesquisa serão derivadas.

1.4 ESTRUTURA DO TRABALHO

O capítulo 2 explica técnicas sistemáticas de projeto de testes, incluindo os critérios de cobertura usados no projeto:

- Critério de particionamento de entrada.
- Critério de grafo.
- Critério de mutação.

[1] https://drive.google.com/file/d/1CFEO6PHUJf5DDRLEZen9vCJ-_X97yM_Z/view?usp=sharing

O capítulo 3 descreve a metodologia aplicada e levanta as principais questões que deveriam ser respondidas pelo trabalho. Em seguida apresenta sucintamente as ferramentas utilizadas, assim como o detalhamento sobre a modelagem dos testes. Finalmente, o capítulo destaca os riscos que podem prejudicar a qualidade das conclusões do trabalho.

O capítulo 4 descreve o conhecimento obtido pela análise estatística. Ele define como será medido o desempenho para cada tipo de teste e aponta os exercícios que apresentaram maior porcentagem de soluções com defeito inédito. O capítulo estuda porque os erros em tempo de execução mais frequentes ocorrem e descreve o tempo médio utilizado para modelar cada critério de cobertura.

O capítulo 5 recapitula todas as descobertas obtidas no decorrer do trabalho e destaca possíveis pesquisas futuras que podem ser implementadas com base neste trabalho.

2 CRITÉRIO DE TESTES DE SOFTWARE

Para compreender esta pesquisa, é preciso definir os termos defeito e falha. Defeitos são quaisquer problemas no programa, que nem sempre geram sintomas. Falhas são manifestações de comportamento incorreto, ou seja, comportamento que não atende aos requisitos do programa. Nesse sentido, os testes, quando falham, indicam a existência de defeitos no código.

Critérios de cobertura facilitam a elaboração de testes e permitem que testes verifiquem de forma direcionada os diversos cenários possíveis (AMMANN; OFFUTT, 2016, p. 56) dentro de programas. A aplicação de critérios diminui redundância nos testes e também o tempo necessário para executá-los nas soluções dos alunos.

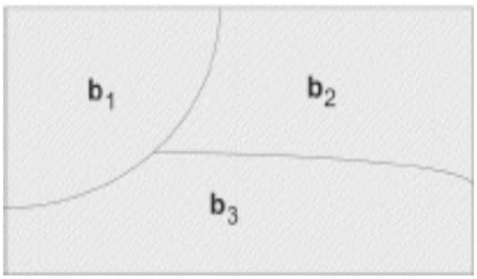
Neste trabalho, os testes foram separados por critério de cobertura para facilitar a comparação da capacidade de identificação de erros. Cada critério é caracterizado por um conjunto de diretrizes que guiam o desenvolvimento dos casos de teste (AMMANN; OFFUTT, 2016, p. 57). Serão analisados 3 critérios de cobertura:

- Particionamento de entrada.
- Grafo.
- Mutação.

2.1 CRITÉRIO DE PARTICIONAMENTO DE ENTRADA

Nesse critério, o domínio de possíveis valores de entrada é separado em vários blocos. O particionamento define em qual bloco ficará cada valor de entrada, onde uma entrada não pode pertencer a mais de 1 bloco do particionamento. No exemplo abaixo, o domínio D é separado em 3 blocos: b_1, b_2 e b_3.

Figura 1 – Particionamento do domínio de entrada D



Fonte: Introduction to Software Testing - Paul Ammann (2008, p. 119)

Esse particionamento é feito com base nas características relevantes do programa. Mas o que poderia ser uma característica relevante? Abaixo estão listados exemplos (AMMANN; OFFUTT, 2016, p. 128):

- Tipo dos parâmetros: quando uma função recebe algum número como entrada, ela pode retornar resultados indesejados se o número for Integer ou Float. Então vale a pena particionar a entrada com um bloco que contém apenas valores Integer e outro para valores Float.
- Valores perto de limites (MAYERS; BADGETT, 2011, p. 83): se um programa media(a, b) calcula a média aritmética de 2 números, vale a pena testar se os parâmetros podem assumir valores positivos, negativos ou 0. Nesse cenário, para pegar números próximos do limite, seria bom testar se 'a' e 'b' podem valer 0, 1 ou -1 sem retornar resultados indesejados.
- Valores com relevância contextual equivalentes: se um programa pode mudar o seu comportamento dependendo do valor da entrada, vale a pena elaborar testes que verifiquem esses valores.

O particionamento visa dividir o domínio de valores da entrada em blocos que contém valores equivalentes, onde todos os valores de um bloco devem se comportar da mesma maneira ao serem testados. Perceba que dentro de um particionamento, nenhuma entrada pode pertencer a mais de um bloco e todas as entradas possíveis encaixam em exatamente 1 bloco de cada particionamento. Para exemplificar esse processo, vamos particionar o domínio para um programa que recebe uma lista não-vazia que contém apenas números inteiros e também um número n inteiro como entrada:

- Particionamento L:
    1. Lista possui apenas números maiores ou iguais a zero.
    2. Lista possui pelo menos um número negativo.
- Particionamento S:
    1. Número n é positivo.
    2. Número n é zero.
    3. Número n é negativo.
- Particionamento P:
    1. Número n está presente na lista.
    2. Número n não está presente na lista.

No nosso projeto, os testes de particionamento de entrada desconsideraram entradas fora do esperado (ou da realidade). Na documentação, cada bloco é identificado pela letra do particionamento seguida por um número. Por exemplo, S1 contém todas as entradas onde n é positivo.

O próximo passo é escolher a estratégia de combinação de blocos, que é o critério de combinação. O critério de combinação define quais requisitos o conjunto de casos de teste vai precisar cobrir (AMMANN; OFFUTT, 2016, p. 133). Nesse trabalho aplicou-se 3 estratégias:

- All Combinations Coverage: Todas as combinações possíveis de todos os blocos, onde toda combinação deve incluir 1 bloco de cada particionamento.
- Pair-Wise Coverage: Todas as combinações possíveis de dois blocos de particionamentos diferentes.
- Each Choice Coverage: Um valor de cada bloco para cada particionamento deve ser utilizado em pelo menos um caso de teste.

Agora vamos exemplificar os requisitos de teste para este particionamento:

- All Combinations Coverage: {(L1, S1, N1), (L1, S1, N2), (L1, S2, N1), (L1, S2, N2), (L1, S3, N1), (L1, S3, N2), (L2, S1, N1), (L2, S1, N2), (L2, S2, N1), (L2, S2, N2), (L2, S3, N1), (L2, S3, N2)}.
- Pair-Wise Coverage: {(L1, S1), (L1, S2), (L1, S3), (L2, S1), (L2, S2), (L2, S3), (L1, N1), (L1, N2), (L2, N1), (L2, N2), (S1, N1), (S1, N2), (S2, N1), (S2, N2), (S3, N1), (S3, N2)}.
- Each Choice Coverage: {L1, L2, S1, S2, S3, N1, N2}.

Todos os requisitos devem ser atendidos por pelo menos um caso de teste. O objetivo dessa prática é garantir que combinações de blocos sejam testadas. Logo, ao definir a estratégia de combinação de blocos, definimos o nível de exigência do projeto de testes. Outro ponto importante sobre os casos de teste é que neste trabalho, optou-se por não testar entradas que quebram as regras do exercício. Por exemplo, um exercício que recebe o tamanho de uma pessoa como entrada não será testado com valores negativos.

Depois de listar os requisitos que devem ser atendidos de acordo com a estratégia de combinação, são elaborados casos de testes que atendem esses requisitos. A maior dificuldade nesta etapa é reduzir a redundância de casos de testes ao cobrir as combinações de requisitos.

O conjunto de casos de teste abaixo satisfaz todos os requisitos de *Pair-Wise* para o particionamento que contém os blocos L1, L2, S1, S2, S3, P1, P2:

- Caso de teste com entrada n = 1 e lista = [1, 2, 3]: satisfaz os requisitos (L1, S1), (L1, P1), (S1, P1).
- Caso de teste com entrada n = 0 e lista = [1, 2, 3]: satisfaz os requisitos (L1, S2), (L1, P2), (S2, P2).
- Caso de teste com entrada n = -1 e lista = [1, 2, 3]: satisfaz os requisitos (L1, S3), (L1, P2), (S3, P2).
- Caso de teste com entrada n = 1 e lista = [-1, -2, -3]: satisfaz os requisitos (L2, S1), (L2, P2), (S1, P2).
- Caso de teste com entrada n = 0 e lista = [0, -1, -2]: satisfaz os requisitos (L2, S2), (L2, P1), (S2, P1).
- Caso de teste com entrada n = -1 e lista = [-1, -2, -3]: satisfaz os requisitos (L2, S3), (L2, P1), (S3, P1).

Finalmente os casos de testes são copiados para o código Python que vai testar as soluções de alunos, junto com a saída esperada para cada caso de teste.

2.2 CRITÉRIO DE GRAFO

Primeiramente, para utilizar o critério de grafo é necessário gerar um artefato de grafo que seja representativo do fluxo dentro do código. Utilizamos então grafos de controle de fluxo (ALLEN, 1970, p. 2-4), que permitem a partir de um código, representar o caminho que uma execução pode tomar, com condicionais gerando bifurcações dentro do grafo. Todo grafo possui um nó inicial e possui um ou mais nós de saída, representando as maneiras que a execução do código pode terminar. Os requisitos de um grafo podem ser definidos pela combinação de nós e arcos a serem explorados (AMMANN; OFFUTT, 2016, p. 159) (ZHU; HALL; MAY, 1997, p. 372-375). Podemos representar a cobertura de um grafo pelas seguintes modalidades:

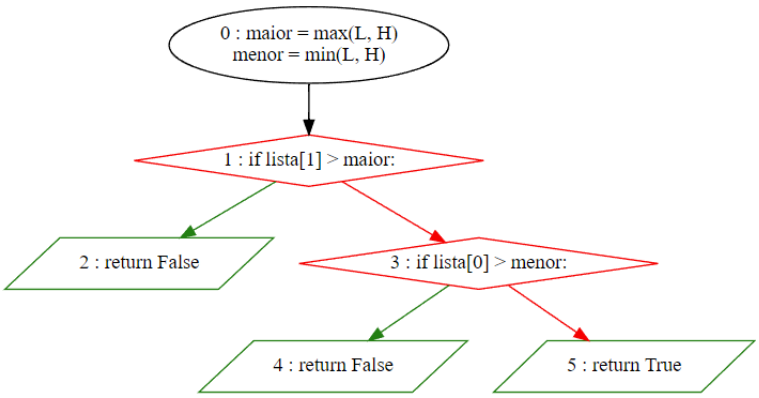
- Cobertura de nós: O conjunto de testes elaborados deve garantir que todo nó tenha sido explorado pelo menos uma vez.
- Cobertura de arcos: O conjunto de testes deve garantir que todo arco presente no grafo seja explorado pelo menos uma vez. Todos os requisitos de arco contém os requisitos de nós.
- Cobertura de par de arcos: O conjunto de testes deve garantir que todo par de arcos consecutivos possível no grafo seja explorado pelo menos uma vez. Assim como no caso anterior, o requisito de par de arcos contém tanto todos os requisitos de nós quanto os requisitos de arcos.

Utilizaremos o problema "Colchão" como exemplo, um dos problemas disponíveis na Machine Teaching. Este problema consiste em propor uma solução para verificar se dadas as proporções de um colchão e de uma porta, determinar se é possível mover o colchão através da porta, com o código gabarito a seguir:

```
def colchao(lista,H,L):
        maior = max(L,H)
        menor = min(L,H)
        if lista[1] > maior:
                return False
        else:
                if lista[0] > menor:
                        return False
                else:
                        return True
```

No código acima as variáveis de entrada são: lista, H e L. lista é uma lista contendo 3 inteiros em ordem crescente representando em centímetros as medidas da cama, H é a altura da porta e L, a largura. A partir deste código, é gerado o seguinte grafo de fluxo:

Figura 2 – Grafo de controle de fluxo do exercício 811



Neste caso, seriam os requisitos antes mencionados definidos como:

- requisitos de nós: {0, 1, 2, 3, 4, 5}
- requisitos de arcos: {[0, 1], [1, 2], [1, 3], [3, 4], [3, 5]}
- requisitos de pares de arcos: {[0, 1, 2], [0, 1, 3], [1, 3, 4], [1, 3, 5]}

Por exemplo, o caso de teste abstrato representado pelo caminho (0,1,2) satisfaz os requisitos de nó 0, 1, e 2, requisitos de arcos [0,1] e [1,2] e requisitos de pares de arcos [0,1,2].

Como o caso de teste (0,1,2) não satisfaz todos os requisitos, o processo deve continuar, levando aos testes abstratos (0,1,3,4) e (0,1,3,5).

Para o teste abstrato (0,1,2), a entrada deve consistir de uma lista cujo o segundo elemento (a segunda maior dimensão do colchão) é maior que a maior medida da porta, retornando False.

Para o teste abstrato (0,1,3,4), esse caminho pode ser cumprido se a segunda maior medida do colchão for menor ou igual à maior medida da porta e se o primeiro elemento (a menor medida do colchão) for maior que a menor medida da porta, retornando False. Finalmente para o teste abstrato (0,1,3,5), as duas menores medidas do colchão devem ambas serem menores que as dimensões da porta, retornando True.

Para propósitos de completude, os testes serão feitos baseados nos requisitos de par de arcos por serem os mais abrangentes e os de maior interesse. Os requisitos de par de arcos permite expor a interação que acontece entre trechos específicos dentro do código.

2.3 CRITÉRIO DE MUTAÇÃO

O critério de mutação utiliza um conceito base diferente para a criação dos casos de teste. O teste de mutação é um critério que se baseia em defeitos típicos que podem acontecer ao se escrever um programa (JIA; HARMAN, 2011) para nos ajudar a criar bons casos de teste. Para isso, são criados diversos mutantes do código-fonte. Um mutante é uma cópia do programa original com uma pequena alteração. Essas mutações, que são feitas através de operadores de mutação bem definidos (funções ou um conjuntos de regras que realizam modificações controladas no código-fonte de um programa, sendo fortemente ligados a linguagem que o código-fonte foi escrito), na grande maioria das vezes, alteram o comportamento do código, e, portanto, a seu resultado final é diferente do que deveria ser. Os casos de teste são criados com o objetivo de distinguir o comportamento do programa original do comportamento do mutante. Se isso ocorrer, o(s) mutante(s) é(são) dito(s) como morto(s), indicando que os casos de teste foram suficientes. Se não ocorrer essa distinção, o(s) mutante(s) sobrevive(m), indicando que os casos de teste utilizados no momento são insuficientes. O objetivo final do teste de mutação é achar um conjunto de testes que matem todos os mutantes criados. Agora, como avaliamos os casos de testes? Quais tipos de operadores existem? Quais são os tipos de mutantes?

Mutation Score:

É a pontuação de um determinado conjunto de casos de teste. É obtido pela razão entre o número de mutantes mortos e o número total de mutantes criados. Quanto maior for essa pontuação, mais efetivo foi aquele conjunto de casos de teste em distinguir o comportamento dos mutantes do código original.

Tipos de Operadores de Mutação (Python - Mutpy):

- AOD (arithmetic operator deletion - Eliminação do Operador Aritmético).
- AOR (arithmetic operator replacement - Substituição do Operador Aritmético).
- ASR (assignment operator replacement - Substituição do Operador de Atribuição).
- BCR (break continue replacement - Substituição do break e continue).
- COD (conditional operator deletion - Eliminação do Operador Condicional).
- COI (conditional operator insertion - Inserção do Operador Condicional).
- CRP (constant replacement - Substituição da Constante).
- DDL (decorator deletion - Eliminação do Decorador).
- EHD (exception handler deletion - Eliminação do Manipulador de Exceção).
- EXS (exception swallowing - Supressão de Exceção).
- IHD (hiding variable deletion - Eliminação da Variável Oculta).
- IOD (overriding method deletion - Eliminação do Método Sobrescrito).
- IOP (overridden method calling position change - Mudança na Posição da Chamada do Método Sobrescrito).
- LCR (logical connector replacement - Substituição do Conector Lógico).
- LOD (logical operator deletion - Eliminação do Operador Lógico).
- LOR (logical operator replacement - Substituição do Operador Lógico).
- ROR (relational operator replacement - Substituição do Operador Relacional).
- SCD (super calling deletion - Eliminação da Chamada de Super).
- SCI (super calling insert - Inserção da Chamada de Super).
- SIR (slice index remove - Remoção do Índice de Fatia).

Para deixar mais claro o que seriam esses operadores de mutação daremos exemplos de operadores (GITHUB...,) usados nos exercícios da Machine Teaching:

- Exemplo 1: AOR em
  `return min(a // 2, b // 3, c // 5)` gera
  `return min(a / 2, b // 3, c // 5)`

- Exemplo 2: COI em
  `if (Cv * 3 + Ce) > (Fv * 3 + Fe)` gera
  `if not ((Cv * 3 + Ce) > (Fv * 3 + Fe))`

- Exemplo 3: ROR em
  `if (quantidadePapel // (competidores * quantidadeFolhas)) == 0` gera
  `if quantidadePapel // (competidores * quantidadeFolhas) != 0:`

- Exemplo 4: LCR em
  `if i < 0 or i >= len(s):` gera
  `if (i < 0 and i >= len(s)):`

- Exemplo 5: SIR em
  `return s[:i] + x + s[i+1:]` gera
  `return (s[:]  + x) + s[i+1:]`

- Exemplo 6: AOD em
  `return str.join(' ',lista[::-1])` gera
  `return str.join(' ', lista[::1])`

Tipos de Mutantes:

- Mutante Morto (mutante foi discriminado com sucesso pelos casos de teste).

- Mutante Sobrevivente (mutante não foi discriminado pelos casos de teste atuais).

- Mutante Incompetente (ocorre quando a modificação no código-fonte gerou um mutante que dispara um erro mesmo antes dos casos de teste rodarem). Os mutantes incompetentes podem ser considerados como mortos.

  Há ainda uma classificação interna dentro dos mutantes sobreviventes, os Mutantes Equivalentes (GRUN; ZELLER, 2009) (mutantes que sobreviveram, mas que o seu comportamento é exatamente igual ao teste original, não sendo possível criar um caso de teste para matá-lo.) Os mutantes equivalentes devem ser desconsiderados na hora de calcular o Mutation Score.

Tendo essas informações em mente, todos os conjuntos de casos de testes foram criados para terem o maior Mutation Score e o menor tamanho referente a quantidade de testes possível. Além disso, como esse critério é baseado em defeitos típicos que podem ocorrer, os testes receberam nomes específicos para orientar os alunos na correção daquele determinado defeito.

3 METODOLOGIA

Para analisarmos a diferença de desempenho entre utilizar um dos critérios rigorosos abordados nesse estudo (critério de mutação, critério de grafo ou critério de particionamento de entrada) para automatizar a criação de testes ou utilizar os testes já implementados na plataforma da Machine Teaching, focamos em responder as 4 perguntas seguintes:

1. Qual o desempenho e o custo-benefício do uso de critérios baseados em particionamento do espaço de entrada para testar as soluções de problemas de introdução à programação?

2. Qual o desempenho e o custo-benefício do uso de critérios baseados em grafos para testar as soluções de problemas de introdução à programação?

3. Qual o desempenho e o custo-benefício do uso de critérios baseados em análise de mutantes para testar as soluções de problemas de introdução à programação?

4. Como essas técnicas se comparam em desempenho com os testes utilizados atualmente pela ferramenta?

Nosso objetivo com essas perguntas foi medir o desempenho de detecção de defeitos e aferir o custo-benefício de cada critério quanto ao seu tempo e dificuldade de implementação nos problemas da Machine Teaching.

Esta pesquisa experimental estuda a mudança de comportamento nos resultados de testes quando se aplica técnicas de testagem de software. Esse procedimento exigiu:

1. Modelagem de testes

2. Implementação de scripts para aplicar os testes em massa

3. Modelagem do banco de dados

4. Implementação de scripts para extrair dados

5. Análise dos dados obtidos

6. Derivar conclusões a partir da análise

Cada um desses procedimentos será melhor detalhado e explicados nas seções seguintes deste texto.

Teoria de testes de software contém conceitos e técnicas que foram fundamentais para guiar a modelagem e implementação dos novos testes. Todos os exercícios e soluções foram

obtidos em plataformas de ensino reais. 33 exercícios distintos foram analisados, cujas soluções enviadas por alunos totalizam mais de 300.000 respostas que foram testadas. Esses dados foram utilizados para o desenvolvimentos de outros testes, estes baseados em critérios diferentes e bem definidos.

As soluções disponibilizadas pela plataforma Machine Teaching são anônimas e não contém identificação pessoal sobre os autores.

Para garantir a qualidade dos testes para cada critério, cada membro dessa pesquisa formulou seus testes individualmente, usando um dos critérios já mencionados nesse estudo, sem interferência dos outros membros. Em seguida, com o propósito de evitar casos de testes incorretos, a entrada e a saída esperada de todos os casos de teste foram revisadas por algum outro autor do estudo. Essa revisão reduz as chances de algum teste retornar classificações errôneas.

Para comparar os resultados, foi necessário coletar os dados com o uso de scripts Python.

Para os três critérios abordados nesse texto, foi necessário estudar a teoria do livro Introduction do Software Testing (AMMANN; OFFUTT, 2016), que fornece o conhecimento para modelagem de testes. A modelagem dos testes para o critério de entrada consistiu nos seguintes passos:

- Definição de características relevantes da entrada do programa
- Escolha da estratégia de combinação
- Selecionar os requisitos a serem atendidos
- Definir os casos de teste que atendem os requisitos

Para o critério de grafo, segue-se os seguintes passos:

- A criação de um grafo de fluxo representativo do gabarito.
- A extração de casos de testes abstratos a partir dos caminhos possíveis do grafo
- interpretação dos casos de testes abstratos para casos de testes concretos

Para o Critério de mutação, os seguintes passos foram necessários:

- Criação de um caso inicial;
- Analisar Relatório dos Mutantes
- Criação de casos de teste para os mutantes sobreviventes até não restar nenhum
- Eliminar testes redundantes

Durante a criação dos testes e a análise dos resultados, foi necessário lidar com certas limitações, tais como:

- Dados Sujos (incompletos ou incorretos)
- Dificuldades no uso das ferramentas Python para criar os testes
- Erros na modelagem dos testes
- Problemas durante a aplicação automatizada de testes

Na análise, foi comparado o resultado de cada teste de maneira abrangente:

- Concordância e Discordância entre os testes originais e os novos
- Diferença de performance entre os testes originais e os novos
- O melhor critério de cobertura para identificar erros
- Custo-benefício da implementação desses novos testes comparados aos testes originais

Esse estudo investiga o comportamento dos dados obtidos na aplicação automatizada dos testes nas soluções de alunos. Os dados foram disponibilizados pela equipe responsável por manter a plataforma Machine Teaching. Com a ajuda de scripts na linguagem Python, os testes de software foram aplicados em 300.000 soluções de estudantes. Será investigado o custo-benefício de cada critério, já que é impossível testar todos os cenários (KANER; PETTICHORD, 2001, p. 6).

3.1 CRITÉRIOS DE AVALIAÇÃO DO DESEMPENHO DOS TESTES

Para definir o que representa desempenho, foi necessário comparar os resultados dos testes. Considerando que o objetivo de testar programas é encontrar defeitos no código, o desempenho será calculado a partir da comparação de 4 tipos de testes para os mesmos problemas. Essa comparação foi feita depois de rodar todos os testes, com o estudo da discordância entre os seguintes testes:

- Testes originais.
- Testes modelados a partir do critério de particionamento de entrada.
- Testes modelados a partir do critério de grafo.
- Testes modelados a partir do critério de mutação.

O estudo da discordância compara a classificação dos testes listados acima, e separa as soluções entre casos de concordância e discordância. A concordância ocorre quando os 4 grupos de teste retornam a mesma classificação para uma solução, enquanto a discordância ocorre quando os 4 grupos de teste acima não retornam a mesma classificação. Se qualquer caso de teste que pertence a uma categoria acima falhar uma solução específica enquanto todos os outros testes passam, temos discordância.

Depois de calcular o desempenho, será analisado o custo-benefício para cada critério. O custo-benefício compara os critérios de cobertura e considera o custo de tempo para modelar testes, o número de casos de teste e o desempenho.

3.2 FERRAMENTAS UTILIZADAS

Para testar as soluções dos alunos e comparar o desempenho dos novos testes com os originais, foi necessário usar um conjunto de ferramentas para cada etapa listada abaixo:

- Modelagem de testes
- Modelagem de banco de dados
- Rodar testes nas soluções
- Extração de dados obtidos na aplicação de testes
- Gerenciamento do banco de dados
- Análise e consulta de dados

A linguagem Python foi escolhida para programar os scripts porque todos os membros são familiares com a linguagem e também porque Python possui várias bibliotecas que atendem as principais necessidades do trabalho.

O código está armazenado no repositório do GitHub [1]. Dentro do repositório, o arquivo README.md contém as instruções para criar o banco de dados, extrair as soluções armazenadas nos arquivos Pickle e aplicar os testes.

Para o projeto dos testes foram usadas ferramentas específicas para cada técnica. Para a técnica baseada em particionamento de equivalência não foram necessárias ferramentas especializadas, apenas apoio à documentação com Google Docs e Google Sheets. Essa documentação está disponível na pasta do Google Drive [2].

A criação dos grafos de fluxo, necessária para o projeto de testes baseado em grafos, foi feita com a biblioteca Py2CFG [3], que permite gerar grafos baseado nos gabaritos providenciados para os exercícios da Machine Teaching.

[1] https://github.com/troclaux/tcc_machine_teaching
[2] https://drive.google.com/drive/folders/16fHDR0o-J1RWlby5uxYhIQPj7IzeJivh?usp=sharing
[3] https://py2cfg.readthedocs.io/en/latest/

Para gerar e executar os mutantes, base para a técnica de projeto de testes baseado em mutação de código, foi usada o MutPy, uma ferramenta de teste de mutação do Python. Essa biblioteca suporta o módulo unittest padrão e gera relatórios YAML/HTML que foram importantes para a análise eficiente dos mutantes e a posterior criação dos casos de teste. (GITHUB...,)

A modelagem do banco de dados foi feita na plataforma DrawSQL [4] por sua simples e intuitiva construção de diagramas entidade relacionamento (ER). O manuseio, inserção e consulta de dados foram feitos com consultas SQL. O gerenciamento do banco de dados foi feito com o MySQL [5], por sua capacidade de rodar comandos SQL. Já o armazenamento e manuseio dos dados extraídos foi feito com a biblioteca SQLAlchemy [6].

Para aplicar testes nas soluções dos alunos, foi escolhida a biblioteca PyTest [7], que oferece várias funcionalidades para testar software.

Para processar o grande número de combinações possíveis de testes e soluções, optou-se pela biblioteca Joblib [8]. Esta ferramenta viabilizou a testagem e extração de dados seja computada paralelamente. Com isso, o tempo para aplicar todos os testes foi reduzido de uma semana para menos de 20 horas.

3.3 MODELAGEM DO BANCO DE DADOS

Primeiramente selecionamos quais seriam as métricas, que serão elaboradas em detalhe na análise estatística. Em seguida, foi feito o planejamento do banco de dados. O modelo precisava armazenar dados sobre cada teste, para permitir a comparação dos critérios de cobertura. Para a entidade *Test*, se armazena os seguintes atributos:

- test_id: Identificador do teste.
- problem_id: Identifica o problema que será verificado pelo teste.
- criteria_type: Tipo de critério de cobertura do teste. Pode assumir os seguintes valores:
  - "input" para critério de particionamento de entrada.
  - "graph" para critério de grafo.
  - "mutation" para critério de mutação.
- pass_count: Contador de soluções que passaram o teste.
- fail_count: Contador de soluções que falharam o teste.

[4] https://drawsql.app/
[5] https://docs.oracle.com/en-us/iaas/mysql-database/doc/getting-started.html
[6] https://www.sqlalchemy.org/
[7] https://docs.pytest.org/en/7.3.x/
[8] https://joblib.readthedocs.io/en/latest/

- error_count: Contador de soluções que retornam erro na execução do teste. Mais especificamente, soluções que retornam qualquer erro que não é *AssertionError*.

Além disso, o banco deveria conter dados sobre cada solução de aluno, para fazer consultas dos resultados de cada caso de teste. Para armazenar essa informação, temos a entidade *Solution*, que representa dados sobre a solução específica de um aluno. Essa entidade contém os seguintes atributos:

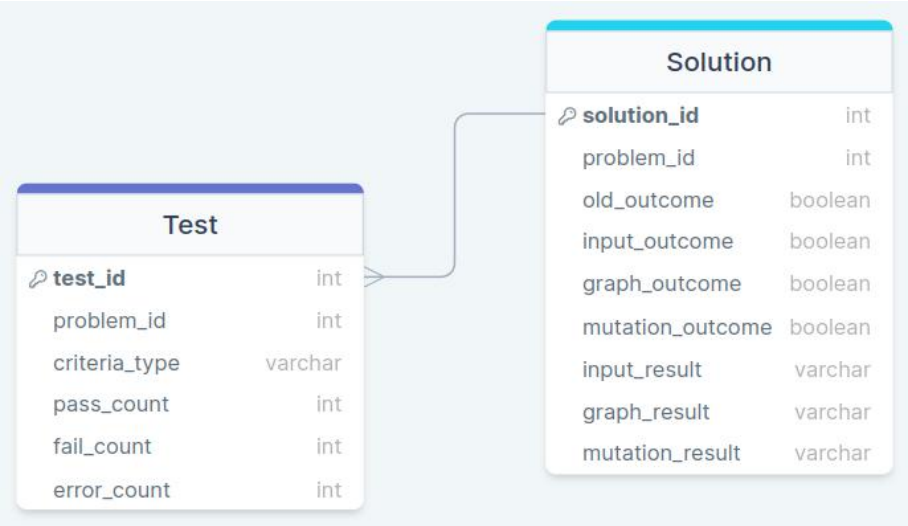
- solution_id: Identificador da solução.
- problem_id: Identifica o problema que a solução atende.
- old_outcome: Resultado (aprovação ou reprovação) do exercício segundo os testes originais do Machine Teaching. Valor booleano que pode ser True ou False.
- input_outcome: Resultado final do exercício para todos os testes de particionamento de entrada. Se 1 ou mais casos de teste falharem ou resultarem em erro, input_outcome terá valor False. Caso contrário terá valor True.
- graph_outcome: Resultado final do exercício para todos os testes de grafo. Se 1 ou mais casos de teste falharem ou resultarem em erro, input_outcome terá valor False. Caso contrário terá valor True.
- mutation_outcome: Resultado final do exercício para todos os testes de mutação. Se 1 ou mais casos de teste falharem ou resultarem em erro, input_outcome terá valor False. Caso contrário terá valor True.
- input_result: String que contém o resultado individual para cada caso de teste de particionamento de entrada. Pode conter os seguintes caracteres:
  - P: solução passou nesse caso de teste.
  - F: solução teve falha de asserção nesse caso de teste.
  - E: solução deu erro nesse caso de teste.
  - T: solução deu timeout nesse caso de teste.
- graph_result: String segue o mesmo padrão que o input_result, só que contém o resultado dos casos de testes pro critério de grafo.
- mutation_result: String segue o mesmo padrão que o input_result, só que contém o resultado dos casos de testes pro critério de mutação.

É importante destacar que a distinção entre falha e erro é utilizada apenas no banco de dados. Na análise estatística, para facilitar a interpretação dos resultados, qualquer

falha ou erro de execução entre os casos de teste de um determinado critério de cobertura serão referidas como falha.

Perceba que os atributos <critério>_outcome consideram todos os casos de teste, enquanto os atributos <critério>_result são strings que contém o resultado individual para cada caso de teste. Por exemplo, quando input_result tem como valor a string "PPFP", significa que a solução passa nos 2 primeiros casos de teste, falha o terceiro e passa o quarto. No final, o modelo do banco de dados está ilustrado na Figura 3:

Figura 3 – Modelo do banco de dados



3.4 METODOLOGIA DO PROJETO DE TESTES

O processo de modelagem de testes foi documentado para facilitar a compreensão e viabilizar quaisquer pesquisas acadêmicas futuras com base neste trabalho. Em seguida, o processo e a documentação para cada critério de cobertura serão descritos.

3.4.1 Particionamento de entrada

A documentação da modelagem para todos os exercícios de particionamento de entrada está nos arquivos do tipo Google Doc, que estão contidos na seguinte pasta do Google Drive [9]. Toda a documentação segue a mesma estrutura, que representa como foi feito o particionamento do domínio de entrada para cada problema:

- Particionamento.
- Critério de combinação.
- Requisitos.

[9] https://drive.google.com/drive/folders/16fHDR0o-J1RWlby5uxYhIQPj7IzeJivh?usp=sharing

- Casos de testes para satisfação de requisitos.

O particionamento é representado em matrizes similares a do quadro 1. A primeira coluna da matriz contém letras que identificam cada particionamento do domínio. Já a primeira linha contém os números que identificam cada bloco do particionamento. Cada bloco é identificado pela combinação da letra com o número. No particionamento do quadro 1, o bloco que contém todas as entradas onde n é negativo é referenciado como bloco B2.

Quadro 1 – Matriz do particionamento do domínio de entrada

(lista, n)	1	2
A	Número n já existe na lista	Número n não existe na lista
B	Número n é igual ou maior que zero	Número n é negativo

A seção "critério de combinação" mostra a estratégia de combinação de requisitos escolhida, que pode ser *Pair-Wise*, *All Combinations* ou *Each Choice*. Em seguida, existe a seção "requisitos", que contém todos os grupos de requisitos que serão satisfeitos pelos casos de teste. A seção "casos de teste para satisfação de requisitos" ilustra quais casos de teste atendem quais combinações de requisitos em exercícios que usam a estratégia de combinação Pair-Wise.

Quando a estratégia de combinação é Pair-Wise, utiliza-se uma matriz para representar quais pares de requisitos são atendidos por quais casos de teste. A posição de índice (1, 1) mostra os parâmetros do programa. A matriz, que sempre é quadrada, contém na primeira linha e coluna a identificação de todos os blocos de todos os particionamentos. Tanto a linha quanto a coluna possuem os mesmos elementos na mesma ordem. Na matriz 4 que segue o particionamento do quadro 1, todos blocos são listados: A1, A2, B1 e B2. As posições restantes serão preenchidas pelos rótulos dos casos de teste que atendem os requisitos daquela posição. Por exemplo, CT1 atende aos requisitos A1 e B1. Isto significa que nesse caso de teste, o número n já existe na lista e que o número n é igual ou maior que zero.

Figura 4 – Matriz Pair-Wise do particionamento do quadro 1

(lista, n)	A1	A2	B1	B2
A1				
A2				
B1	CT1	CT3		
B2	CT2	CT4		

Acima temos uma matriz que ilustra a estratégia *Pair-Wise* para um problema hipotético que recebe uma lista não-vazia de números "lista_numero" e um número inteiro "n"como entrada. Foram definidos os seguintes casos de teste para esse particionamento do quadro 1:

- CT1: ([1, 2, 3], 1) satisfaz os requisitos (A1, B1)
- CT2: ([-2, -1, 3, 4], -1) satisfaz os requisitos (A1, B2)
- CT3: ([1, 2, 5], 4) satisfaz os requisitos (A2, B1)
- CT4: ([-2, 1, 5], -3) satisfaz os requisitos (A2, B2)

Cada caso de teste tem o valor da entrada definido e recebe um número de identificação. Por exemplo, na figura 4 o caso de teste com entrada ([-2, -1, 3, 4], -1) é identificado como CT2 e na mesma linha indica-se quais pares de requisitos são atendidos pelo CT2. A partir disso os índices da matriz correspondentes aos pares de requisitos são preenchidos com o rótulo do casos de teste correspondente.

Blocos que começam com a mesma letra pertencem à mesma partição. É impossível ter elementos que combinam dois blocos do mesmo particionamento, já que os blocos de um mesmo particionamento são conjuntos disjuntos. Por conta disso, todos as posições "inválidas"que ocupam 2 blocos do mesmo particionamento estão com fundo preto, como (A1, A2) ou (B2, B1). Para evitar repetição, o preenchimento dos casos de testes em uma coluna ocorre apenas abaixo dos blocos com fundo preto. Finalmente, se múltiplos casos de teste cobrem uma combinação de requisitos, os números que identificam os casos de testes serão separados por barras. Exemplificando, se CT2, CT5 e CT10 atendem simultaneamente 2 blocos, a representação na matriz será CT2/5/10.

3.4.2 Grafo

Para a metodologia de grafos, foi optada a criação de testes que cumprem com cobertura total dos pares de arcos no grafo gerado a partir do gabarito de cada exercício. Os grafos são gerados pela biblioteca py2cfg, cujo proposito é gerar um grafo de controle de fluxo a partir de um código. Em seguida, é reestruturado o objeto resultante para uma lista com as informações pertinentes que serão utilizadas para recolher informações do grafo. O nó 0 é sempre o nó raiz do grafo gerado.

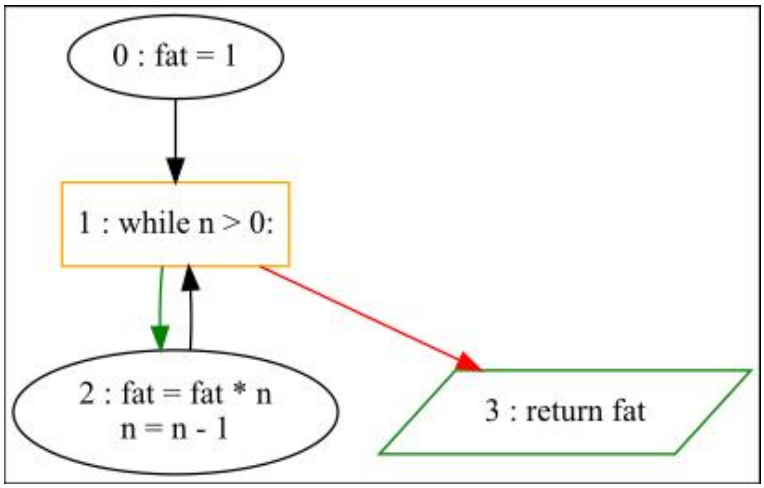
Os testes abstratos são gerados utilizando uma busca em profundidade invertida, utilizando cada nó final do grafo como um ponto de partida para a busca. A cada nó explorado, o algoritmo busca um nó pai que tenha menor índice primeiro, pois isso garante que o caminho gerado eventualmente chegará ao nó 0, o nó raiz. Uma vez que o primeira iteração da busca completa e chega no nó 0, é registrado o caminho percorrido e os requisitos que foram satisfeitos naquela iteração. Feito isto, o algoritmo continua sua

execução de onde parou, tentando buscar caminhos alternativos para a raiz. Existe um limite de profundidade para evitar que o algoritmo fique preso em caminhos circulares que podem estar presentes no grafo. Se o limite é atingido, o algoritmo considera como se tivesse atingido um final e começa a percorrer caminhos alternativos. Caso não consiga, é reiniciado o algoritmo com um limite dobrado.

Se ao chegar no nó raiz e não tiver mais requisitos a serem satisfeitos, é então retornado a lista de testes abstratos, que então são interpretados para testes concretos, utilizando entradas que repliquem um padrão similar aos caminhos definidos pelos testes abstratos. Para definirmos os casos concretos de testes, devemos observar os casos de testes abstratos e entender qual a propriedade que o caminho do teste abstrato busca testar.

Utilizando o exercício 821 (fatorial), que consiste em desenvolver uma solução que dado um número n, retorne o valor fatorial de n, temos o seguinte grafo de controle de fluxo:

Figura 5 – Grafo de controle de fluxo do exercício 821



Nesse exemplo, temos os seguintes requisitos de par de arcos: {[0, 1, 2], [0, 1, 3], [1, 3, 4], [1, 3, 5]}. Seriam criados os caminhos [0, 1, 2], [0, 1, 3, 4] e [0, 1, 3, 5]. O algoritmo inicia sua busca no nó 3, subindo em direção ao nó 0. Uma vez exausto os caminhos alcançados a partir do nó 3, ele mudaria seu inicio para o próximo nó de return caso tivesse, até exaustar todos os caminhos possíveis.

Seguindo o exemplo do problema 821, o podemos interpretar os casos de teste como:

- CT1: [0, 1, 3] é satisfeito quando a entrada n é igual a 0.
- CT2: [0, 1, 2, 1, 3] acontece quando temos apenas uma repetição. Especificamente testando o laço de repetição do while com os requisitos [0, 1, 2], [1, 2, 1], [2, 1, 3]. Uma caso de teste concreto onde n é igual a 1 satisfaz CT2.

- CT3: [0, 1, 2, 1, 2, 1, 3] especificamente testa o requisito [2, 1, 2], quando ocorre mais de uma repetição do laço do while. Qualquer caso de teste concreto cuja entrada n é maior que 1 satisfaz CT3.

3.4.3 Mutação

Para a metodologia de mutação, optamos por sempre obter a pontuação de mutação (mutation score) máxima tentando desenvolver o menor número de testes possível. Isso é para averiguar qual a efetividade da cobertura proporcionada pelo critério de mutação quando é usado o número mínimo de testes para isso. Não foi feita nenhuma seleção prévia de operadores de mutação nem o uso de operadores experimentais, ou seja, trabalhamos com todos os operadores que a ferramenta do MutPy, na configuração padrão, disponibilizava. Esses operadores estão descritos na seção 2.1.3 deste texto.

Os arquivos de códigos analisados são os gabaritos dos exercícios passados para os alunos no Machine Teaching. O arquivo contendo esse código será referenciado como "gabarito.py". Para usar a ferramenta do MutPy é necessário, também, ter um arquivo com, pelo menos, um caso de teste, para que biblioteca funcione e gere os mutantes. O arquivo contendo os casos de teste será referenciado como "testes.py".

Com os arquivos gabarito.py e testes.py preenchidos corretamente, para gerar os mutantes é usado o seguinte comando: 'mut.py –target gabarito.py –unit-test testes.py'. Com isso, a ferramenta MutPy vai gerar um relatório no terminal detalhando as mutações geradas a partir do gabarito.py. Além disso, é possível também gerar um relatório html adicionando o trecho '-m –report-html relatorio_html_gabarito' ao final do comando anterior. Esses relatórios do MutPy contém cada um dos mutantes, os operadores de mutação que foram utilizados para criá-los, uma indicação de qual caso de teste matou aquele determinado mutante e uma relação de todos os mutantes mortos, sobreviventes, incompetentes e que geraram um time-out na execução. Com base nesse relatório, é possível determinar quais mutações foram detectadas com êxito (mutantes mortos) e quais não foram (mutantes sobreviventes). A partir disso, o próximo passo é analisar sobre quais operadores os mutantes que sobreviveram atuam e criar casos de testes cujo resultado final seja alterado pela mutação sofrida nesse operador.

Para melhor elucidar, considere uma função qualquer "def foo(a)" que, originalmente retorna a/2. Ou seja, ela retorna o valor da variável "a" dividido por 2, sendo essa uma operação que vai retornar um float. Ao gerar os mutantes dessa função, o operador aritmético "/" sofrerá mutação e, uma das possíveis alterações que ele poderá passar será a troca para "//", que realiza uma divisão inteira. Digamos que esse seja o único mutante gerado. Se no caso de teste for passado a = 4, o mutante vai sobreviver, visto que $4/2 = 4//2$ (2.0 = 2). Entretanto, caso seja passado a = 5, o mutante será morto, visto que $5/2$!= $5//2$ (2.5 != 2).

O processo descrito a cima é o mesmo usado para analisar todos os outros arquivos do tipo gabarito.py vindos do Machine Teaching. Os passos gerais são (GEEKSFORGEEKS,):

- Instalar a ferramenta MutPy.
- Escrever um caso de teste simples para o MutPy funcionar.
- Gerar os mutantes usando o comando "mut.py –target gabarito.py –unit-test testes.py".
- Analisar o relatório detalhado gerado pela ferramenta.
- Identificar os mutantes sobreviventes.
- Analisar os Operadores de Mutação utilizados.
- Identificar os mutantes equivalentes e desconsidera-los para a pontuação de mutação final.
- Construir casos de teste que o resultado final seja diferente entre o do mutante e o do código original.
- Repetir até se esgotarem os mutantes sobreviventes.
- Voltar aos casos de teste, procurar e eliminar redundâncias.

Todos os gabaritos, scripts python e relatórios html estão disponíveis no anexo A.

4 ANÁLISE

Nesta análise, o desempenho de cada teste será medido pela quantidade de defeitos que ele identifica. Como todos os casos de testes foram conferidos por múltiplas pessoas, todo teste que resultar em falha será interpretado como evidência da presença de algum defeito na solução.

Para medir a eficácia dos testes novos, vamos utilizar o termo "defeito inédito". Uma solução com defeito inédito ocorre quando uma solução que foi aprovada pelos testes originais reprova qualquer caso de teste dos novos testes. O motivo para usar essa métrica está na seguinte linha de raciocínio: se uma solução passa os testes originais, mas falha qualquer teste novo, esta solução provavelmente apresenta um defeito que não foi detectado originalmente.

Todos os gráficos gerados no decorrer deste trabalho estão armazenados neste documento [1]. Ele contém dois tipos de gráficos: *heatmaps* e gráficos de pizza. Esses gráficos ilustram a distribuição de soluções em cada cenário para todos os exercícios. Cada solução testada possui um resultado agregado para os seguintes grupos de teste:

- Testes originais (usados atualmente pela ferramenta na correção das atividades dos alunos).
- Testes modelados a partir do critério de particionamento de entrada.
- Testes modelados a partir do critério de grafo.
- Testes modelados a partir do critério de mutação.

Cada item acima terá seu próprio resultado agregado, que combina todos os seus respectivos casos de teste. Em um critério qualquer, se 1 ou mais casos de teste resultarem em erro de execução ou falha, o resultado agregado desse critério falha ("F"), caso contrário o resultado agregado passa ("P"). Por exemplo, se todos os testes modelados a partir do critério de mutação passarem, o resultado agregado do critério de mutação passa também. Por outro lado, se qualquer um dos casos de teste falhar ou resultar em erro de execução, o resultado agregado de mutação vira uma falha.

Os resultados agregados para os 4 grupos listados acima vão encaixar em 1 dos 16 cenários possíveis no quadro 2. Cada cenário é representado por 4 letras, cada letra pode assumir o valor "P" ou "F". A primeira letra sempre é o resultado agregado dos testes originais, a segunda é o resultado agregado dos testes que foram modelados conforme o critério de particionamento de entrada. A terceira letra faz o mesmo para critério de

[1] https://docs.google.com/document/d/1tPUoon9xo0OiQNQ2lgrp-kV9t-85NGncW09ldaKu4Mc/edit?usp=sharing

grafo e a quarta representa o mesmo para o critério de mutação. Exemplificando, se uma solução em algum exercício encaixar no cenário PFPF, esta solução:

- Passa todos os testes originais.
- Falha pelo menos 1 caso de teste elaborado a partir do critério de particionamento de entrada.
- Passa todos os testes elaborados a partir do critério de grafo.
- Falha pelo menos 1 caso de teste elaborado a partir do critério de mutação.

Quadro 2 – Cenários

FFFF	FFFP	FFPF	FPFF
FFPP	FPFP	FPPF	FPPP
PFFF	PFFP	PFPF	PPFF
PFPP	PPFP	PPPF	PPPP

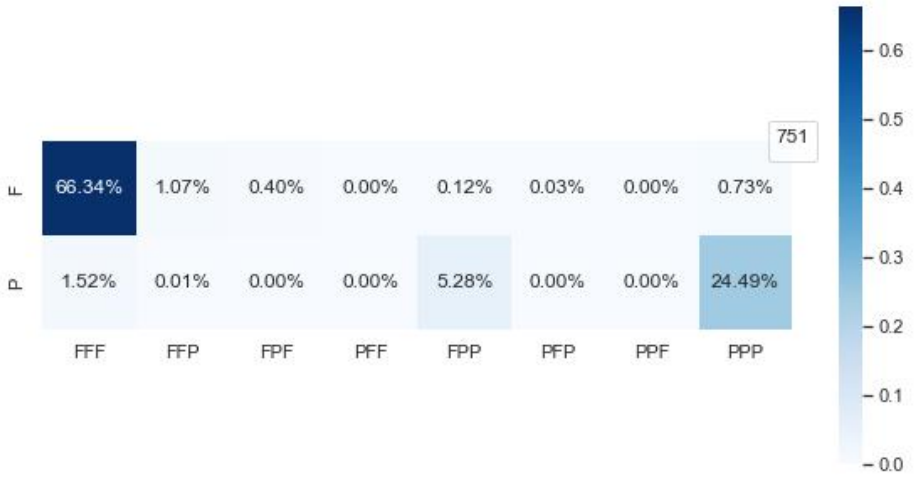
Além dos cenários ilustrados no quadro 2, existe outro jeito de separar as soluções. Elas podem ser categorizadas em 3 casos, conforme é possível observar na figura 7:

- Caso de concordância: Quando todos os 4 resultados agregados são iguais.
  - Todas as soluções que pertencem aos cenários FFFF e PPPP.
- Caso de discordância por falta de cobertura (tipo 1): Os cenários pertencentes a esta categoria são as soluções que foram reprovadas pelos testes da Machine Teaching (testes originais), mas que foram aprovados por pelo menos 1 critério. Esse tipo de discordância significa que os casos de testes do critério que aprova foram insuficientes para identificar algum defeito.
  - Todas as soluções que pertencem aos cenários: FFFP, FFPF, FPFF, FFPP, FPFP, FPPF e FPPP.
- Caso de discordância por defeito inédito (tipo 2): Contém todos os cenários que foram aprovados pelos testes originais e que reprovaram pelo menos 1 teste novo. Poderíamos considerar esses casos como defeitos inéditos identificados pelos novos testes.
  - Todas as soluções que pertencem aos cenários: PFFF, PFFP, PFPF, PPFF, PFPP, PPFP e PPPF.

Todos os gráficos estão presentes na seguinte pasta do Google Drive [2]. O quadro 2 contém todos os cenários possíveis que estão presentes nos *heatmaps*. No *heatmap* do exercício 811 (o problema do colchão, visto anteriormente) o cenário PFFP contém 7,64% das soluções, isso significa que os testes dos critérios de particionamento de entrada e grafo provavelmente contém algum defeito inédito.

Abaixo temos o heatmap do exercício 751, que pede para o aluno implementar uma função que conta o número de palavras em uma frase. Ele apresenta concordância de todos os testes para 90,83% das soluções (soma de todas soluções classificadas como FFFF e PPPP). Adicionalmente, 5,28% de soluções no caso de discordância PFPP sugerem a existência de defeito inédito. Outro provável defeito inédito está presente no cenário PFFF, que contém a pequena porcentagem de 1,52% das soluções. O restante está distribuído nos outros cenários, em pequenas proporções.

Figura 6 – Heatmap do exercício 751

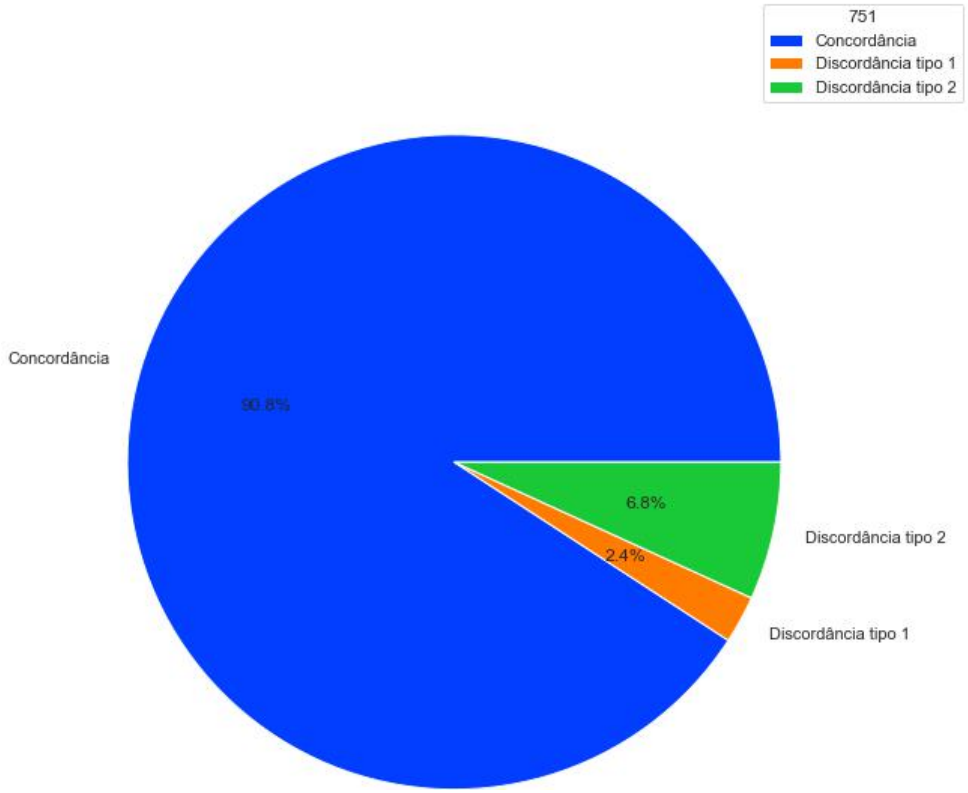


Fonte: Classificação das soluções do exercício 751

Logo em seguida se observa o gráfico de pizza do exercício 751 novamente, este gráfico é composto por 3 partes: concordância, discordância tipo 1 (falta de cobertura) e discordância tipo 2 (defeito inédito). Esse gráfico ilustra de forma mais intuitiva o consenso entre os testes originais e os antigos, assim como falta de cobertura e a identificação de defeitos inéditos, com as porcentagens arredondadas para uma casa decimal. Aproximadamente 90,8% dos casos apresentam concordância, 6,8% das soluções foram classificadas como discordância tipo 2 e 2,4% como discordância tipo 1.

[2] https://docs.google.com/document/d/1tPUoon9xo0OiQNQ2lgrp-kV9t-85NGncW09ldaKu4Mc/edit?usp=sharing

Figura 7 – Gráfico de pizza do exercício 751



Fonte: Classificação das soluções do exercício 751

No quadro 3 existem 5 colunas, que contém as seguintes informações:

- Problema: número identificador do problema.
- Soluções: quantidade de soluções que foram enviadas por alunos na plataforma Machine Teaching.
- Testes originais: número de casos de testes que a plataforma Machine Teaching possui originalmente.
- Testes particionamento: número de casos de teste que foram modelados a partir do critério de particionamento de entrada.
- Quantidade de testes por grafo: número de casos de teste que foram modelados a partir do critério de grafo.
- Quantidade de testes por mutação: número de casos de teste que foram modelados a partir do critério de mutação.

Quadro 3 – Quantidades de soluções e casos de teste por problema

Problema	Soluções	Testes originais	Testes particionamento	Testes grafo	Testes mutação
736	6706	22	5	1	1
744	10403	101	4	1	1
751	6721	5	4	1	1
798	5973	5	4	4	1
800	4738	10	2	6	1
804	26687	12	6	8	1
806	13821	5	4	4	5
807	21940	24	2	1	3
809	10092	10	9	1	1
810	20115	10	2	1	1
811	10400	10	9	3	3
812	13761	10	2	1	1
815	8689	10	6	1	1
816	12510	10	5	1	1
817	14486	10	9	1	1
819	11342	20	6	4	1
820	13620	10	6	4	2
821	6730	9	6	2	1
822	7920	16	3	3	1
823	10395	11	9	2	2
824	15247	10	6	3	1
827	8019	10	4	3	1
828	8491	20	6	6	3
829	5414	10	6	3	1
831	9174	10	4	5	1
832	10474	21	5	2	2
833	7580	12	11	3	1
834	5650	10	8	2	1
835	10078	4	3	1	1
836	10177	3	4	5	1
838	7151	10	3	1	1
839	16370	10	4	1	1
840	9205	10	11	1	3

O quadro 4 descreve a distribuição dos casos de concordância e discordância. A coluna quantidade de soluções representa o total de soluções recebidas para cada problema, onde cada solução será classificada como caso de concordância (todos os testes retornam o mesmo resultado - FFFF ou PPPP) ou caso de discordância tipo 1 (os testes originais identificaram defeitos que os novos não identificaram) ou caso de discordância tipo 2 (os testes novos identificaram defeitos que os originais não identificaram). Por exemplo, o problema 751 recebeu 6.721 soluções enviadas por alunos como resposta. Dessas 6.721 (100%) soluções, 6.105 (90,83%) foram classificadas como caso de concordância. Ape-

nas 2 casas decimais estão visíveis, já que valores a partir da terceira casa decimal são arredondados para baixo.

Quadro 4 – Concordância e discordância para cada problema

Problema	Quantidade de soluções	Soluções com concordância (%)	Soluções com discordância tipo 1 (%)	Soluções com discordância tipo 2 (%)
736	6706	98,66	0,84	0,51
744	10403	96,09	2,98	0,93
751	6721	90,83	2,35	6,81
798	5973	89,27	5,04	5,69
800	4738	78,73	10,57	10,7
804	26687	84,5	7,79	7,7
806	13821	86,97	7,97	5,06
807	21940	82,3	17,16	0,53
809	10092	97,39	1,36	1,25
810	20115	85,08	11,54	3,39
811	10400	66,4	22,73	10,87
812	13761	89,88	4,88	5,25
815	8689	97,86	1,16	0,98
816	12510	79,43	10,22	10,34
817	14486	74,33	21,77	3,9
819	11342	97,25	1,32	1,43
820	13620	89,26	7,28	3,47
821	6730	90,74	2,02	7,24
822	7920	89,53	3,38	7,08
823	10395	87,75	7,86	4,39
824	15247	87,8	10,51	1,69
827	8019	89,57	7,33	3,09
828	8491	84,49	2,04	13,47
829	5414	86,57	8,09	5,34
831	9174	74,06	12,94	13,0
832	10474	96,26	2,31	1,43
833	7580	88,59	9,47	1,94
834	5650	89,95	7,68	2,37
835	10078	92,01	4,95	3,04
836	10177	87,56	1,97	10,47
838	7151	92,11	6,96	0,92
839	16370	89,46	10,18	0,35
840	9205	90,7	8,48	0,81

4.1 COMPARAÇÃO DO DESEMPENHO DOS TESTES

Para identificar o critério com melhor desempenho, bastou analisar o *heatmap* de cada exercício e contar em quantos exercícios cada critério teve discordância com resultado agregado "F". Por exemplo, no exercício 811 7,64% das soluções caíram no cenário

PFFP. Nesse trabalho nenhum critério (particionamento de entrada, grafo e mutação) individualmente identifica mais soluções defeituosas que o resultado agregado dos testes originais, mas os testes novos agregados identificam mais soluções defeituosas que os testes originais.

Entre as 360.079 soluções, os testes originais identificaram 303.264 soluções com defeitos. Já os testes novos identificam 316.126 defeitos, uma melhoria de aproximadamente 4,2%. O diagrama de Venn da figura 8 representa quantos defeitos foram identificados por cada tipo de teste.

Através de todos os 33 exercícios analisados, a porcentagem média para identificação de defeitos novos foi aproximadamente 4,76%. Além disso, a variância da porcentagem de defeitos novos identificados nesses 33 exercícios foi aproximadamente 0,00296.

Nos diagramas das figuras 8 e 9, *old* corresponde aos defeitos identificados pelos testes originais, *input* corresponde aos defeitos identificados pelos testes que seguem o critério de particionamento de entrada, *graph* corresponde aos defeitos identificados pelo critério de grafo e *mutation* corresponde aos defeitos identificados pelo critério de mutação.

É importante lembrar que o primeiro diagrama ilustra o número de defeitos totais para cada tipo de teste (original, particionamento de entrada, grafo e mutação), enquanto o segundo diagrama ilustra o número de defeitos inéditos identificados pelos testes novos (particionamento de entrada, grafo e mutação). Defeitos inéditos são defeitos que não foram identificados pelos testes originais e foram identificados por algum teste novo.

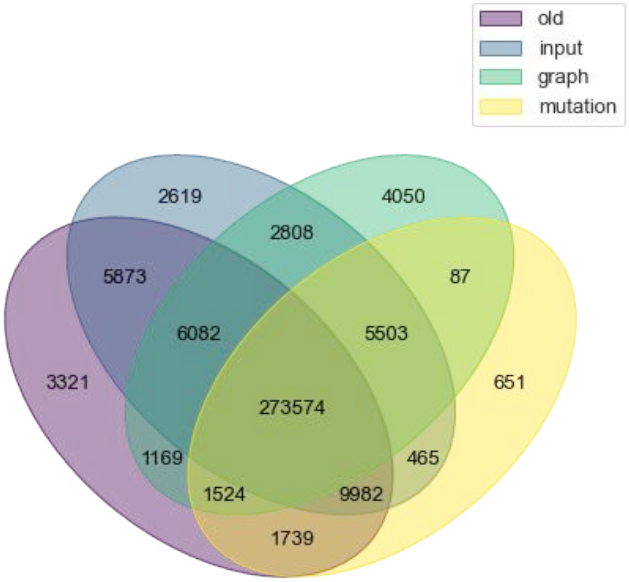
Comparando a capacidade de identificação de defeitos inéditos por critério, foram obtidos os seguintes resultados:

- Testes de particionamento de entrada identificaram 11.395 soluções com defeitos inéditos.
- Testes de grafo identificaram 12.448 soluções com defeitos inéditos.
- Testes de mutação identificaram 6.706 soluções com defeitos inéditos.

Lembrando que um mesmo defeito inédito pode ser identificado por mais de um critério de cobertura simultaneamente, então na lista acima uma mesma solução pode ser contabilizada mais de uma vez.

No geral, o critério de grafo demonstrou ter maior capacidade de identificação de defeitos inéditos. Esse fato foi inesperado, já que o critério de grafo possui menos casos de teste em comparação ao critério de particionamento de entrada. Além disso, nos exercícios iniciais, a simplicidade da resolução limita a elaboração de casos de testes para o critério de grafo. O critério de mutação foi o menos eficaz entre os 3 critérios.

Figura 8 – Defeitos identificados por cada critério de cobertura



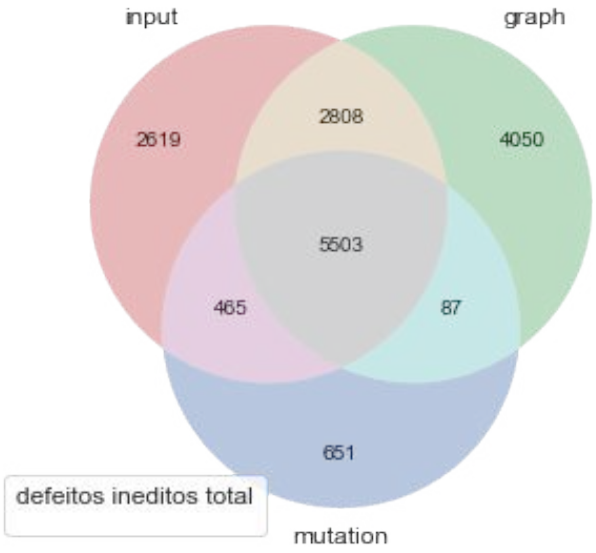
4.2 DEFEITOS QUE NÃO FORAM IDENTIFICADOS POR TESTES NOVOS

É importante analisar também quantos defeitos foram identificados pelos testes originais e não foram identificados por nenhum dos testes novos. Esse fenômeno corresponde ao cenário FPPP, onde a solução falha algum teste original, mas passa por todos os testes novos. A soma total de soluções que foram classificadas como FPPP em todos os exercícios é de 3321 soluções. Comparando os defeitos que não foram identificados pelos testes novos com as 16.183 soluções com defeitos inéditos, fica evidente o aumento de identificação de defeitos inéditos pelos novos testes. Os testes originais identificam mais defeitos que testes novos apenas nos exercícios 824 e 838, onde respectivamente 8,28% e 4,06% das soluções estavam no cenário FPPP. Em todos os outros exercícios, os testes novos identificaram mais defeitos inéditos que os testes originais.

Ao analisar os casos de teste originais sobre cada exercício, percebe-se que muitos defeitos inéditos ocorreram porque testes novos verificavam valores próximos de limite. Valores próximos do limite normalmente são os menores e maiores valores que uma entrada pode assumir, como:

- Números no limite da troca de sinal, por exemplo: 0, 1 e -1

Figura 9 – Defeitos inéditos identificados por cada critério de cobertura



- Strings vazias
- Vetores vazios

A causa desse padrão provavelmente vem do fato que muitos casos de teste dos testes originais selecionavam a entrada de forma aleatória. Por conta dessa seleção aleatória para os casos de testes, muitos valores perto do limite não eram atingidos pelos testes. Essa seleção aleatória de casos de teste ocorreu porque na plataforma Machine Teaching, o instrutor que insere o problema tem liberdade para escolher os casos de teste e ele nem sempre tem a técnica de um testador. Nos testes novos, esses valores receberam mais atenção.

4.3 EXERCÍCIOS COM ALTA PORCENTAGEM DE DEFEITOS INÉDITOS

A média das porcentagens de soluções com defeito inédito entre todos os exercícios é 4,7%. Considerando isso, exercícios com certas características apresentaram porcentagem de soluções com defeito inédito maior que essa média. Na análise dos resultados, alguns tipos de exercício apresentaram porcentagem de identificação de defeitos inéditos maior que a média.

Os 5 exercícios com maior porcentagem de identificação de defeitos inéditos foram:

- Exercício 836: 10,4% das soluções com defeito inédito.

- Exercício 800: 10,7% das soluções com defeito inédito.
- Exercício 811: 10,9% das soluções com defeito inédito.
- Exercício 831: 13% das soluções com defeito inédito.
- Exercício 828: 13,5% das soluções com defeito inédito.

Exercícios exigindo manipulações de string com palavras em português tiveram maior taxa de descoberta de defeitos inéditos. Por exemplo, o exercício 831 pedia para o aluno elaborar uma função chamada lingua_p, que recebe como parâmetro uma palavra em português e retorna essa palavra com inserções da letra 'p' após cada vogal. Esse foi o segundo exercício com maior porcentagem de soluções com defeito inédito, com aproximadamente 13% das soluções apresentando defeito inédito. Outros exercícios de manipulação de string que apresentaram taxa de identificação de defeitos inéditos acima da média foram os exercícios 751, 812 e 820.

Um dos exercícios que teve maior taxa de defeitos inéditos foi o exercício 800. Nesse exercício, o programa deve receber uma lista de strings, que são os produtos que serão comprados, e um dicionário Python que tem como chave o nome do produto e como valor o preço de cada produto. A grande causa de falhas nas soluções ocorreu por conta de erros de sintaxe ao declarar a estrutura de dicionários no código Python. Por exemplo, muitos alunos esqueciam das chaves ou aspas ao declarar o dicionário no programa. Sendo assim, o erro mais frequente nas soluções com defeitos inéditos acabou sendo o *SyntaxError*. Outros erros frequentes que envolviam sintaxe da linguagem Python foram:

- Uso inconsistente das teclas tab e espaço na indentação do código. Em Python, os blocos são definidos pela indentação do código. Desse modo, para o programa funcionar corretamente, é preciso manter consistência ao indentar cada bloco do programa. Muitas vezes o aluno alternava entre indentar com tabs e espaços, o que gerava erros no programa.
- Fechamento incorreto de aspas para string. Muitas vezes os alunos começavam uma string com aspas simples e fechavam com aspas duplas.
- Uso excessivo de aspas delimitar comentários no código. Muitas vezes uma solução começava um comentário com 3 aspas duplas e fechava ele com 4 aspas duplas.

4.4 SOLUÇÕES COM ERRO EM TEMPO DE EXECUÇÃO

Os erros mais frequentes das soluções são contados e exibidos na plataforma Machine Teaching para qualquer conta de administrador. A plataforma mostra o tipo de erro mais frequente para cada "capítulo", que contém entre 3 e 7 exercícios. O erro mais frequente

em todos os capítulos (exceto o último) foi o *SyntaxError*. O segundo erro mais frequente por capítulo variava entre NameError e TypeError. Segue abaixo uma explicação breve desses erros frequentes (Python Software Foundation, ; MATTHES, 2019):

- NameError: ocorre quando uma variável não existe no escopo local ou global.
- SyntaxError: ocorre quando o interpretador encontra sintaxe inválida no código.
- TypeError: ocorre quando uma função ou operação é aplicada em um objeto com tipo incorreto.

Nas soluções, foram identificados problemas frequentes causados pela sintaxe incorreta da linguagem, tais como:

- Uso inconsistente de tabs e espaços para indentar o código: na linguagem Python, os blocos são definidos pela indentação do código. Desse modo, para o programa funcionar corretamente, é preciso manter consistência ao indentar cada bloco do programa. Muitas soluções alternavam entre tabs e espaços na indentação, gerando erros no programa.
- Fechamento incorreto de aspas para strings: o código 1 é um exemplo real que foi enviado na plataforma com esse problema, onde a última linha contém uma string que é fechada incorretamente.
- Uso excessivo de aspas para delimitar comentários no código. Muitas vezes uma solução começava um comentário com 3 aspas duplas e fechava ele com 4 aspas duplas, o que gerava erro de sintaxe.

Durante a análise dos erros frequentes, outro problema foi identificado: a plataforma erroneamente aprova soluções que deveriam apresentar erros em tempo de execução. Esse raro evento é inesperado porque todo teste deveria identificar que a execução do programa foi interrompida. Por exemplo, o código 1 gera um SyntaxError por fechar incorretamente as aspas da string na última linha. Ainda assim, a plataforma aprova o código 1.

Para tentar identificar a origem do erro, consideramos a hipótese onde a causa do problema vem da extração incorreta de dados, aprovando incorretamente soluções com erro em tempo de execução. Para testar essa hipótese, alguns desses programas (incorretamente aprovados) foram enviados e avaliados na plataforma. Ainda assim, essas soluções com SyntaxError foram aprovados pela plataforma. Ao testar soluções com os testes originais diretamente no Bash, os testes originais pararam de aprovar incorretamente soluções com erro, confirmando que a causa da aprovação incorreta de soluções vem da plataforma.

Código 1 – Exemplo de código Python com SyntaxError

```
# Coloque um comentario dizendo o que a funcao faz
# Escolha nomes elucidativos para suas variaveis
# str-> str
def hashtag(s):
    """ recebe e retorna uma string s inserindo  # no inicio, no meio e
       no final"""
    meio=int(len(s)/2)
    inicio=str(s[:meio])
    final=str(s[meio:])
    return str('#' + inicio + '#' + final + '#")
```

4.5 TEMPO MÉDIO DE MODELAGEM PARA CADA CRITÉRIO

Para comparar cada critério, é preciso considerar o custo de tempo para modelar cada caso de teste. Elaborar o tempo utilizado em cada etapa do processo de modelagem pode ajudar equipes de desenvolvimento ao escolher quais tipos de testes devem ser priorizados. Para comparar o investimento de tempo por critério, será elaborado:

- Mudanças no processo de modelagem que reduziram o custo de tempo
- Tempo médio utilizado para cada etapa do processo de modelagem
- Tempo médio total gasto na modelagem de cada exercício

4.5.1 Critério de particionamento de entrada

A elaboração de testes para o critério de particionamento de entrada segue as seguintes etapas:

- Particionamento
- Critério de combinação
- Requisitos
- Casos de teste para satisfação de requisitos

O tempo necessário para modelar depende de vários fatores: complexidade do exercício, particionamento, critério de combinação e da automatização de etapas da modelagem por meio de programas. Esses fatores devem ser considerados ao escolher como priorizar o desenvolvimento de testes.

Os exercícios iniciais eram mais simples, e consequentemente exigiam menos tempo. O processo de modelagem dos testes foi acelerando conforme o membro responsável pelo

critério aprendia. Por exemplo, o tempo necessário para escrever as combinações de requisitos reduziu dramaticamente depois que se desenvolveu um programa que gera todas as combinações de blocos caso a modelagem use *Pair-Wise coverage*.

Antes de modelar o teste, era preciso ler o enunciado do exercício. Depois ocorre a decisão sobre quais serão os particionamentos, que demora aproximadamente 5 minutos. Essa etapa determina o custo de tempo para as etapas seguintes. A escolha do critério de combinação também exigia pouco tempo. Os requisitos demoram pouco tempo também.

A etapa dos casos de teste para satisfação de requisitos foi mais longa, com duração média de 30 minutos. Ela depende do número de particionamentos e do critério de combinação de requisitos. Casos de testes eram adicionados para atender todas as combinações de requisitos definidas na etapa anterior. Cada escolha tomava pouco tempo, mas a soma de todas as escolhas aumentava a duração da tarefa.

No total, demorou 23,8 horas para concluir a modelagem do critério de particionamento de entrada para os 34 exercícios. Desse modo, o tempo médio gasto para cada exercício para fazer a modelagem do particionamento de entrada foi 42 minutos. Foram feitos, em média, 6 casos de teste por exercício.

4.5.2 Critério de grafo

A modelagem dos testes foi feita com base no gabarito de cada exercício, utilizando o PY2CFG para obter a árvore de controle de fluxo do gabarito. A partir da árvore gerada, foram determinados a mão os testes para satisfazer os requisitos de pares de arcos. Enquanto o tempo de criação de testes para exercícios com árvores mais simples era ainda relativamente curto, um pequeno aumento no uso de condicionais no código, implicando um aumento de complexidade na arvore, aumentava muito o tempo para a criação dos testes. Para alguns exercícios, demorava por volta de 40 minutos para desenhar todos os testes que satisfaziam os requisitos de par de arcos de apenas um exercício.

Devido ao alto número de exercícios, tornou-se atraente a possibilidade de automatizar parte do processo de modelagem utilizando a metodologia explicada na seção 3.5.2. A maioria do tempo para criação dos testes era dedicado à interpretação dos testes abstratos que a árvore gerava, em seguida interpretando esses testes abstratos para testes concretos que satisfaziam o caminho gerado pelo teste abstrato. A implementação dessa metodologia encurtou muito o tempo de criação de testes, pois é necessário apenas verificar que os testes abstratos gerados pelo programa possam ser satisfeitos e interpretar o caminho. No pior dos casos, a criação de testes demorava um pouco mais de 8 minutos por exercício. No caso em que algum teste abstrato fosse impossível de realizar, verificava-se se os requisitos que ele satisfazia eram atendidos por outros testes. Caso não fossem, era necessário criar manualmente um teste abstrato que satisfizesse os requisitos que sobraram do teste que não pode ser realizado.

Em casos em que não havia condicionais presentes no gabarito, o grafo gerado era um grafo simples de 2 nós com uma aresta conectando entre eles. Neste caso, qualquer entrada escolhida resultaria em cobertura completa do critério. Com a metodologia aqui apresentada, demorou um pouco menos de 5 horas para completar a criação de testes para os 33 exercícios.

4.5.3 Critério de mutação

A criação dos casos de testes utilizando o critério de mutação respeitava uma série de passos que se repetia para cada um dos exercícios. Esses passos eram:

- Criação de caso de teste genérico. Para gerar o primeiro relatório contendo os mutantes do código original, a biblioteca do MutPy precisa que, pelo menos, haja um caso de teste implementado. O tempo médio para a criação desse caso de teste genérico é de, no máximo, 2 minutos.

- Análise do relatório do MutPy com os mutantes gerados. O relatório do MutPy contém os mutantes ainda sobreviventes, bem como os operadores de mutação que foram utilizados pela ferramenta MutPy para criá-los. É necessário entender como o operador de mutação alterou o funcionamento do código original para pensar em um caso de teste específico que discrimine o resultado final diferente do mutante. Essa é a parte mais demorada de todo o processo de criação de testes utilizando o critério de mutação. Leva em torno de 3 minutos para os casos mais simples e cerca de 15 minutos para os mais complexos. A média total ficou por volta de 7 minutos.

- Criação de um caso de teste que consiga discriminar o mutante do código original, matando o mutante. Após a compreensão de como o operador de mutação alterou o funcionamento do código original, a elaboração e implementação do caso de teste novo pode ser feita. Essa é a parte manual de criação do caso de teste e leva cerca de 3 minutos.

- Repetir os dois últimos passos até matar todos os mutantes. Diferentes códigos podem ter um número diferente de operadores de mutações e, portanto, um número diferente de mutantes. Para exemplificar, 55 foi o maior número de mutantes gerados a partir de um único código. O menor número de mutantes gerados foi 0. Entretanto, mesmo com o aumento de mutantes, o tempo para elaborar todos os casos de teste não aumentou de maneira proporcional. A medida que reaplicamos os dois últimos passos a cada mutante sobrevivente, frequentemente, conseguimos matar mais de 1 mutante por caso de teste, o que acabava por economizar bastante tempo.

- Ao final ocorre uma reavaliação de todos os casos de teste com o intuito de eliminar casos de teste desnecessários. A remoção desses casos de teste redundantes não

é obrigatória para a análise de cobertura do critério de mutação, foi apenas uma escolha metodológica com o fim de descobrir o menor número de casos de teste que cobrisse a maior pontuação de mutação.

No início da pesquisa e da criação dos casos de teste utilizando o critério de mutação, não foi possível extrair completamente todo o potencial de criação rápida de testes a partir do relatório gerado pelo MutPy. Isso se deve pela ainda falta de conhecimento sobre a ferramenta do membro da pesquisa que ficou responsável por esse critério. Após alguns dias de familiarização e utilização da biblioteca, os casos de teste foram feitos de maneira bem mais assertiva e eficiente, alcançando a média descrita a cima. O total de problemas abordados foi de 35. O total de mutantes para todos esses problemas foi de 277. O total de casos de teste criados foi de 96. Realizando todos os cálculos, temos 10,05 horas no total.

4.6 RISCOS

Pesquisas experimentais sempre envolvem riscos e incertezas que podem prejudicar a veracidade das conclusões. A análise estatística possui alguns desses riscos listados abaixo:

- Escolha das métricas de desempenho: a escolha inapropriada das métricas que serão armazenadas no banco de dados pode prejudicar a análise estatística e gerar conclusões imprecisas ou pouco confiáveis.

  As métricas utilizadas para essa avaliação foram: identificação de defeitos, quantidade de casos de teste e esforço para o projeto dos casos de teste. Essas métricas não necessariamente capturam toda informação necessária para avaliar efetivamente o desempenho de diferentes conjuntos de testes.

- Viés de amostragem: como todos os exercícios e soluções foram obtidos de disciplina introdutória de programação, é possível que os dados não representem o desempenho dessa estratégia de modelagem de testes para disciplinas de programação mais avançadas. Esse risco foi contornado com a conclusão sendo limitada a esse escopo.

- Extração de soluções a partir do arquivo no formato Pickle: a extração incorreta dos arquivos Pickle pode alterar o conteúdo das soluções. Essas alterações podem causar erros no programa quando ele for testado. Um exemplo disso seria qualquer alteração na indentação do código, que pode gerar erros críticos ao rodar a solução.

- Os resultados de aprovação ou reprovação de cada solução foram obtidos por caminhos diferentes para os testes originais e para os testes baseados em critérios. Os testes originais foram avaliados através da plataforma Machine Teaching que "corrige" involuntariamente alguns dos defeitos sintáticos. Sendo assim, os testes originais nem tiveram a oportunidade de identificar esses defeitos.

- A plataforma Machine Teaching está em constante evolução, portanto, é importante ressaltar que os testes analisados podem ter sido modificados durante o desenvolvimento deste trabalho.

5 CONCLUSÃO

Este trabalho extraiu e analisou dados visando medir o desempenho e o custo-benefício de cada tipo de teste. Para medir esses atributos, comparamos os testes novos e os testes originais em vários aspectos. Primeiramente o desempenho dos 3 critérios foi comparado, onde o desempenho é medido a partir da capacidade de identificação de defeitos inéditos. Ao comparar os 3 critérios, percebe-se que o critério de grafo identificou mais defeitos inéditos. Em segundo lugar veio o critério de particionamento de entrada, seguido pelo critério de mutação em último lugar. O desempenho dos testes que seguiram o critério de grafo foi inesperado já que eles, em média, contém menos casos de teste que o critério de particionamento de entrada. Ainda assim, o critério de grafo demonstrou o melhor desempenho em relação aos outros 2 critérios.

A próxima questão a ser respondida seria o custo-benefício de cada critério. O custo-benefício leva em consideração o tempo médio necessário para projetar os testes, assim como a dificuldade de implementar testes nos problemas da Machine Teaching. 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.

A aplicação de testes para avaliar soluções na plataforma Machine Teaching permitiu estudar as diferenças de resultados entre os testes originais e testes novos com uso de dados reais. No geral, os testes novos (juntando os 3 critérios) aumentaram a identificação de defeitos em aproximadamente 4,2% em relação aos testes originais. Esse número, no entanto, foi provavelmente influenciado pelo fato da plataforma corrigir alguns defeitos antes do teste. Um novo experimento onde os testes originais são executados fora da plataforma, no mesmo arcabouço dos testes baseados em critério precisa então ser realizado para verificação da influência desse comportamento da Machine Teaching.

5.1 TRABALHOS FUTUROS

A possibilidade de construir trabalhos futuros com base nessa trabalho é promissora porque todo o arcabouço está disponível nos links do anexo A. O arcabouço contém os links para o código-fonte, documentação, dados e instruções para reproduzir o experimento. A partir disso, existem as seguintes possibilidades para trabalhos futuros.

5.1.1 Testar soluções de alunos de outra instituição e com novos problemas

Aplicar a estratégia de modelagem com critérios de cobertura em outro ambiente. Comparar os resultados deste trabalho com resultados obtidos em outra plataforma ou

com alunos de outra instituição de ensino e novos problemas pode gerar melhor entendimento sobre o efeito da aplicação de teoria de testes para avaliação de estudantes.

5.1.2 Estudo sobre os erros mais frequentes nas soluções

Durante a análise dos exercícios com maiores porcentagens de soluções com defeitos inéditos, percebe-se que o erro mais frequente foi o erro de sintaxe (SyntaxError). Essa informação pode auxiliar os instrutores da disciplina de programação para focar no uso correto da sintaxe da linguagem de programação durante o ensino de disciplinas de programação.

5.1.3 Uso de testes em disciplinas de programação mais avançadas

Aplicar modelagem de testes com critérios de cobertura em disciplinas de programação mais avançadas. Neste trabalho, são testados apenas exercícios de programação que foram aplicados em disciplinas de programação para iniciantes.

REFERÊNCIAS

ALLEN, F. E. Control flow analysis. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 5, n. 7, p. 1–19, jul 1970. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/390013.808479.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 2. ed. [S.l.]: Cambridge University Press, 2016. ISBN 9781107172012.

BAKER, R. S. Stupid tutoring systems, intelligent humans. **International Journal of Artificial Intelligence in Education**, v. 26, p. 600–614, 2016. ISSN 1560-4292,1560-4306. Disponível em: http://doi.org/10.1007/s40593-016-0105-0.

EVERETT, G. D.; MCLEOD, R. **Software Testing: Testing Across the Entire Software Development Life Cycle**. 1. ed. [S.l.]: John Wiley and Sons, 2007. ISBN 047179371X.

GEEKSFORGEEKS. **Mutation Testing using Mutpy Module in Python**. Último acesso em 28/06/2023. Disponível em: https://www.geeksforgeeks.org/mutation-testing-using-mutpy-module-in-python/.

GITHUB MutPy. Último acesso em 28/06/2023. Disponível em: https://github.com/mutpy/mutpy.

GRUN, D. S. B. J. M.; ZELLER, A. The impact of equivalent mutants. 2009. Disponível em: https://www.st.cs.uni-saarland.de/publications/files/gruen-mutation-2009.pdf.

IBM. **What is Software Testing and How Does it Work? | IBM**. Último acesso em 30/05/2023. Disponível em: https://www.ibm.com/topics/software-testing.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, p. 649–678, 2011. Disponível em: http://doi.org/10.1007/s40593-016-0105-0.

KANER, J. B. C.; PETTICHORD, B. **Lessons Learned in Software Testing**. 1. ed. [S.l.]: John Wiley and Sons, 2001. ISBN 9780471081128.

MATTHES, E. **Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming**. No Starch Press, 2019. ISBN 9781593279288. Disponível em: https://books.google.com.br/books?id=nkh8tgEACAAJ.

MAYERS, C. S. G.; BADGETT, T. **The Art of Software Testing**. 3. ed. [S.l.]: John Wiley and Sons, 2011. ISBN 1118031962.

MITROVIC, A.; OHLSSON, S. Evaluation of a constraint-based tutor for a database language. University of Canterbury. Computer Science and Software Engineering., 1999.

MORAES, L. **Machine Teaching**. Último acesso em 15/04/2023. Disponível em: http://machineteaching.tech/en/.

MORAES, L. O.; PEDREIRA, C. E. Designing an intelligent tutoring system across multiple classes. In: **CSEDM@EDM**. [S.l.: s.n.], 2020.

Python Software Foundation. **Python Documentation - Built-in Exceptions**. Último acesso em 02/07/2023. Disponível em: https://docs.python.org/3/library/exceptions.html.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, p. 366–427, dec 1997. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/267580.267590.

ANEXOS

ANEXO A – LINKS PARA MATERIAL DO PROJETO

1. Todos os problemas que foram analisados estão disponíveis no seguinte link:
   https://github.com/troclaux/tcc-machine-teaching/tree/main/enunciados

2. Todos os gabaritos para os problemas acima estão disponíveis no seguinte link:
   https://drive.google.com/drive/folders/1bXQLqm0aV29l558e5tpHRkCJRfixzFeb?usp=sharing

3. Todas as soluções estão armazenadas nos arquivos Pickle, acessíveis por meio do link:
   https://drive.google.com/file/d/1CFEO6PHUJf5DDRLEZen9vCJ-_X97yM_Z/view?usp=sharing

4. Os documentos com as modelagens de todos os testes para todos os critérios estão disponíveis no seguinte link:
   https://drive.google.com/drive/folders/16fHDR0o-J1RWlby5uxYhIQPj7IzeJivh?usp=sharing

   - Os documentos cujo nome começa com "Modelagem particionamento"contém a modelagem para todos os testes que seguem o critério de particionamento de entrada. As tabelas utilizadas nesses documentos estão nas planilhas cujo nome começa com "Tabelas particionamento".
   - O arquivo cujo nome começa com "Modelagem grafo"contém a modelagem para todos os testes que seguem o critério de grafo.

5. Os gráficos de pizza gerados no decorrer deste trabalho estão disponíveis no seguinte link:
   https://docs.google.com/document/d/1tPUoon9xo0OiQNQ2lgrp-kV9t-85NGncW09ldaKu4Mc/edit?usp=sharing

6. Instruções para reproduzir o experimento estão no arquivo README.md do repositório GitHub:
   https://github.com/troclaux/tcc-machine-teaching#readme

7. Todo o banco de dados está disponível dentro dos arquivos CSV contidos na seguinte pasta do Google Drive:
   https://drive.google.com/drive/folders/1M7oTlRLVKUMpWY2lIRdU8_9d5Ku3XQV6?usp=sharing

8. Todos os scripts, casos de testes e relatórios html referentes ao critério de mutação estão na seguinte pasta do Google Drive:
   https://drive.google.com/drive/folders/1y2PiMmV3CraSYMtZs4XEsG4j2E2yU1q2

9. Todos os diagramas de Venn que representam visualmente os defeitos identificados por cada tipo de teste estão na seguinte pasta do Google Drive:
   https://drive.google.com/drive/folders/1C3Xgf_9RUR9igkPfNhXcklbbgvJnCeVX?usp=sharing