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

CÉSAR AUGUSTO JULIO DA SILVA

Testes Baseados em Particionamento da Entrada para Plataforma de Ensino de Programação.

RIO DE JANEIRO
2024

CÉSAR AUGUSTO JULIO DA SILVA

Testes Baseados em Particionamento da Entrada para Plataforma de Ensino 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
2024

CIP - Catalogação na Publicação

S421t

Silva, César Augusto Julio da
Testes baseados em particionamento da entrada para plataforma de ensino de programação. / César Augusto Julio da Silva. -- Rio de Janeiro, 2024.
52 f.

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

1. Teste de software. 2. Particionamento de entrada. 3. Plataforma de ensino de programação. I. Moreira, Anamaria Martins, orient. II. Título.

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

CÉSAR AUGUSTO JULIO DA SILVA

Testes Baseados em Particionamento da Entrada para Plataforma de Ensino 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 17 de julho de 2024

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
ANAMARIA MARTINS MOREIRA
Data: 20/08/2024 08:09:21-0300
Verifique em https://validar.iti.gov.br

Anamaria Martins Moreira
Doutora (Universidade Federal Do Rio de Janeiro)

gov.br Documento assinado digitalmente
DANIEL CHICAYBAN BASTOS
Data: 20/08/2024 20:20:53-0300
Verifique em https://validar.iti.gov.br

Daniel Chicayban Bastos
Doutor (Universidade Federal Do Rio de Janeiro)

gov.br Documento assinado digitalmente
LAURA DE OLIVEIRA FERNANDES MORAES
Data: 21/08/2024 21:27:21-0300
Verifique em https://validar.iti.gov.br

Laura de Oliveira Fernandes Moraes
Doutora (Universidade Federal Do Estado Do Rio De Janeiro)

Dedico esse trabalho à minha família e meus amigos que me apoiaram durante toda a minha trajetória de vida.

AGRADECIMENTOS

Expresso aqui meus agradecimentos à professora Anamaria, por me orientar ao longo do projeto e providenciar com dicas, comentários e críticas que me ajudaram a tornar esse trabalho possível. Agradeço aos colegas do projeto "MODELAGEM DE TESTES DE SOFTWARE", em particular François Boéchat, cujo projeto não só me possibilitou a ideia para esse documento, mas também respondendo as minhas dúvidas e questões. Agradeço também à equipe da Machine Teaching pela disponibilização dos dados necessários e ao pessoal do LC3, em particular ao Bruno Alves, por disponibilizar acesso ao maquinário e me auxiliar com a configuração do computador. Agradeço a meus amigos, em particular Rafael Guzmán e Marco Terra, que me motivaram a seguir em frente. Por último, agradeço aos meus pais, Jurandir e Roseli, que continuam a me apoiar até hoje.

RESUMO

Ao longo da última década, ocorreu uma expansão na área de desenvolvimento de plataformas de ensino online. Dentre essas, uma que foi desenvolvida e está em uso por alunos da UFRJ é a Machine Teaching, focada no ensino de programação. Nela são disponibilizados problemas de programação em conjunto com seus respectivos testes, o que permite uma avaliação mais detalhada do desempenho dos alunos, automatização do processo de correção e um retorno imediato dos resultados. Um projeto final de curso recente focou em realizar uma comparação entre os testes já implementados pelos professores com testes modelados seguindo metodologias mais formais e consolidadas na área. Dentre essas, uma das metodologias aplicadas foi a de particionamento do espaço de entrada. O método de particionamento do espaço de entrada visa uma análise baseada em requisitos voltados para a entrada do programa, os quais os casos de testes tentam cobrir segundo um critério de cobertura determinado. Tal método permite uma aplicação em caixa-preta, onde o criador dos testes somente tem acesso às especificações do projeto. Uma segunda vantagem é que a justificativa de cada caso de teste depende somente das características das entradas do programa. Esse trabalho tem como objetivo uma nova análise dos particionamentos criados durante o projeto acima mencionado de forma a corroborar os seus resultados e a criação de um pequeno manual de recomendações para escrita dos enunciados e projeto de testes. O alcance de tais resultados foi obtido por meio de: obtenção dos problemas, soluções e códigos dos alunos, realização de uma nova análise dos exercícios usando método de particionamento do espaço de entrada, execução dos testes criados em conjunto com os da plataforma e os do projeto anterior, e extração e comparação dos dados obtidos. O projeto encontrou uma porcentagem mais alta de falhas nos testes baseado em particionamento de entrada do que os produzidos na plataforma em 28 dos 35 exercícios, com uma média de 3.82% e uma mediana de 1.31% de códigos que falharam somente nos testes novos em comparação a uma média de 1.00% e uma mediana de 0.16% que só falharam nos testes da Machine Teaching. Além disso, por meio de uma análise mais detalhada dos códigos dos alunos em conjunto com as falhas mais comuns em cada problema, foi criado um guia de dicas para criação de enunciado cujo objetivo é ajudar a evitar tais erros e uma lista de dicas para projeto de teste, com o objetivo de auxiliar na criação de casos de testes com uma cobertura melhor.

Palavras-chave: teste de software; particionamento de entrada; critério de cobertura; plataforma de ensino de programação.

ABSTRACT

In the last decade, there was an expansion on the area of development of online learning platforms. Between those, one that was developed and is being used by students of Federal University of Rio de Janeiro is Machine Teaching, which is focused on teaching programming. In it, programming problems are available together with its respective test cases, which allow for a more detailed assessment of the students' performance, automatic correction and immediate return of the results. A group of students did their final project focused on making a comparison between the tests already implemented on the platform and tests modeled using more formal and consolidated methodologies in the area. One of those methods was the input space partitioning. The method of input space partitioning aims at an analysis based on requisites of program input, whose test cases try to cover a determined coverage criterion. Such method allows a black-box application, where the creator of the tests only has access to the program's specification. A second advantage is that each justification for each test case depends on the input domain. This project has the goal of making a new analysis of the partitions created during the aforementioned project to corroborate the results and to create a small guide of tips to improve the problems' descriptions e the tests' creation. These results were obtained by: getting the problems, solutions and the students codes, making a new analysis of the problems descriptions using the input partitioning method for test creation, executing the tests together with the ones from the platform and created by the previous group, extraction and comparison of the data obtained. The project found a higher percentage of failures on tests created by this project than the ones on the platform in 28 of the 35 problems, with an average of 3.82% and a median of 1.31% on codes that failed only in the new tests in comparison to an average of 1.00% and a median of 0.16% that failed only on the Machine Teaching tests. Besides that, the defects found were made available as a guide to lower the points of ambiguity in the transmission of the goal to be reached by the student, possibly avoiding common errors. A list of tips to use during the tests' creation was also made available, based on the defects and experiences during this project, with a goal to help create tests with a better coverage.

Keywords: software test; input partitioning; coverage criteria; programming education platform.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas, ocorreu um grande aumento nas tecnologias de transmissão de dados, facilitando o acesso a Internet e, com isso, a comunicação remota entre pessoas. Essa comunicação permite não só o diálogo à distância mas também assíncrono, removendo muitas das dificuldades associadas a problemas de locomoção ou duração. Com essa facilidade, a sociedade viu uma maneira de aumentar sua eficiência se conseguisse implementar tal acessibilidade em todas as áreas de produção e serviços possíveis. A área de ensino é uma dessas áreas cujo o acesso remoto tem sido implementado (COATES; JAMES; BALDWIN, 2005). Apesar do acesso remoto fornecer um amplo leque de funcionalidades que podem ser implementadas dentro dessa área, esse projeto foca especificamente em plataformas de ensino remoto de programação. As plataformas de ensino remoto são ambientes de aprendizagem cujo propósito é apoiar a transmissão de conhecimento ao registrar cursos, exercícios, usuários, assim permitindo a administração e reutilização do conteúdo, podendo até registrar informações sobre os usuários permitindo uma análise aprofundada de suas experiências de aprendizagem (OLIVEIRA; CUNHA; NAKAYAMA, 2016). Em 2018, na UFRJ, um ambiente de aprendizagem online chamado Machine Teaching foi implementado como ferramenta de apoio em cursos introdutórios de programação. Suas funções incluem auxiliar os alunos em suas práticas de programação, os professores na correção dos exercícios, e, sobretudo, coletar dados sobre o conhecimento dos alunos durante o processo de aprendizagem em programação. Tais dados são analisados e usados para apoiar tomadas de decisões relativas ao curso e ao aprendizado de programação (MORAES et al., 2022).

A plataforma Machine Teaching possui como objetivo o aprendizado de construção de programas legíveis e modulares em Python. Ela enfatiza a resolução de problemas, decomposição em funções (modularidade) e domínio de habilidades básicas. A dinâmica de interação entre aluno e a plataforma é feita por códigos modulares que devem ser definidos de forma a receber uma quantidade de parâmetros e emitir uma resposta. Com isso, a avaliação é feita por meio de um conjunto de testes criado pelo professor, os quais contêm diferentes entradas e suas correspondentes respostas esperadas, automatizando a correção e permitindo uma resposta imediata do sistema para o aluno. Então, a correção das atividades é fortemente dependente da qualidade do conjunto de testes utilizados. Conjuntos de testes de baixa qualidade podem levar o aluno a concluir que seu código está correto quando, na realidade, está defeituoso.

No ano de 2023, um grupo de alunos do curso de ciência da computação apresentou um trabalho de conclusão de curso (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023) analisando uma amostra de exercícios publicados na plataforma Machine Teaching em conjunto com os códigos submetidos pelos alunos como soluções para os problemas e o re-

sultado dessas submissões quando executadas na plataforma contra os testes cadastrados nela. Tal análise focou em utilizar técnicas de teste de software já estabelecidas e conhecidas na área de software para criar novos conjuntos de testes, executando os códigos dos alunos contra esses conjuntos e comparando esses resultados obtidos com os resultados recebidos da plataforma para averiguar quanta discrepância ou concordância existe entre os resultados desses novos conjuntos e os resultados recebidos pela plataforma. Esses dados foram fornecidos pela equipe da Machine Teaching e consistem em todas as submissões feitas por alunos durante diversos períodos letivos, totalizando 386025 códigos. Esses novos conjuntos de testes foram criados utilizando três tipos de critérios de cobertura (AMMAN; OFFUTT, 2017): particionamento de entrada, grafo e mutação. Os critérios de cobertura utilizados focam em estabelecer requisitos para os testes, cada um tendo regras específicas de criação de requisitos (AMMAN; OFFUTT, 2017). O experimento mostrou que dentre os três tipos o que teve melhor desempenho foi o de grafos, seguido pelo particionamento de entrada e depois mutação.

Dos critérios no trabalho acima mencionado, o critério de particionamento de entrada é o foco desse trabalho. Esse tipo de critério de cobertura é recomendado quando a pessoa quer estabelecer requisitos que sejam baseados nos valores de entrada do programa e a especificação do programa. Na Machine Teaching, essa especificação é o enunciado do problema. Um fator de possível interesse desse critério para a Machine Teaching é a facilidade de mapear os casos de teste a itens do enunciado em comparação a outros critérios, podendo ser usado no futuro como apoio adicional ao aluno. Além disso, revisitar esse critério vai permitir explorar defeitos comuns entre enunciados e casos de testes que podem ser usados para atualizar casos de testes já na plataforma ou quando criados em novos exercícios.

1.1 OBJETIVO

O objetivo desse projeto é fazer uma nova avaliação do critério de particionamento de entrada para a mesma amostra de exercícios que o grupo de alunos coletou, criando novos requisitos e casos de testes a fim de atestar os resultados alcançados. Além disso, a partir dos resultados, criar um guia com dicas que auxiliem o professor que está criando novos problemas na plataforma em diminuir ambiguidade no enunciado, ajudando a transmitir o objetivo do exercício ao aluno. Por último, criar uma lista de dicas para o projeto de teste, de forma a auxiliar o projetista a criar casos de testes com uma boa cobertura dos possíveis defeitos.

1.2 ESTRUTURA DO TRABALHO

O capítulo 2 traz a fundamentação do projeto, explicando conceitos necessários para a leitura do relatório como: projeto de testes, requisitos, particionamento de entrada,

critério de cobertura, além de explicar sobre a plataforma Machine Teaching e o projeto feito pelo grupo anterior (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023). O capítulo 3 relata a metodologia, falando sobre todos os passos de execução do projeto, incluindo os programas utilizados. O capítulo 4 mostra os resultados obtidos dos experimentos e suas análises. O capítulo 5 contém um guia de projeto de enunciados e uma lista de dicas de projeto de teste, ambos conseguidos por meio da análise feita no capítulo 4. O capítulo 6 traz a conclusão obtida durante o projeto em conjunto com a sugestão de alguns trabalhos futuros.

2 FUNDAMENTAÇÃO

Antes de entrar na metodologia do projeto, é necessário explicar um pouco mais sobre a base desse projeto: a Machine Teaching, o critério de particionamento de entrada, e o projeto anteriormente realizado que analisa o resultado dos testes implementados na plataforma e o uso de metodologias baseadas em critérios.

2.1 MACHINE TEACHING

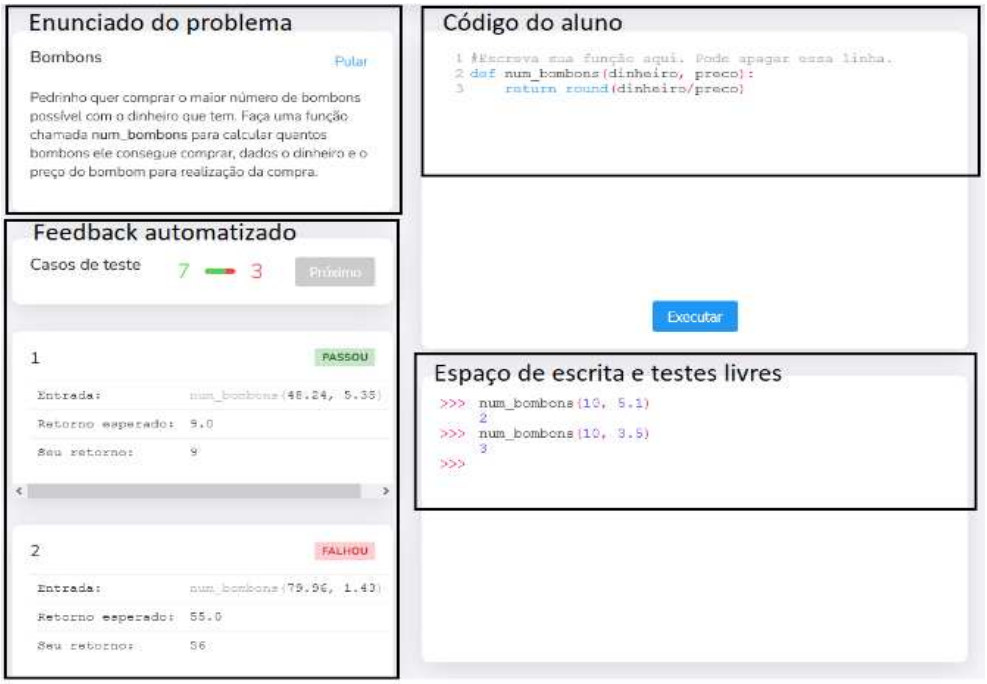
Como dito anteriormente, a Machine Teaching é um ambiente de aprendizado online cuja função é apoiar as aulas práticas dos cursos introdutórios de programação oferecidos pelo Instituto de Computação da UFRJ via a coleta de dados sobre os exercícios dos alunos durante as aulas, e, com tais dados, tomar decisões sobre o método de ensino. Esse é o principal diferencial da plataforma em relação a outros ambientes de ensino. Além disso, seus objetivos secundários são: fornecer retorno imediato durante as atividades e automatizar a análise dos códigos dos alunos, facilitando a interação aluno e professor. Outra vantagem é a praticidade que o aluno tem da plataforma manter seu código disponível e a possibilidade de acessar os exercícios de sua casa ou qualquer outro lugar via o acesso à internet (MORAES et al., 2022).

Na plataforma, a resolução dos problemas ocorre em uma interface composta pelas seguintes janelas (Figura 1): uma janela para desenvolvimento do código, uma janela para escrita de testes, uma para o enunciado do problema, e uma para resposta da execução do código contra os testes elaborados, fornecendo todas as informações necessárias para o aluno conseguir compreender o problema oferecido e desenvolver uma solução necessária. Os testes podem ser criados ou por meio de uma função que compõe um caso de teste de forma aleatória, assim fornecendo testes aleatórios, ou que os testes sejam escritos diretamente na plataforma.

O aluno também tem acesso a informações com relação ao seu progresso nas listas, tempo de resolução, individual e comparado à turma, e quantidade de erros (Figura 2). A Machine Teaching também disponibiliza aos professores dados relativos às turmas de forma que o mesmo pode averiguar tempo de resolução dos exercícios, erros mais comuns, e o progresso geral do alunos, como mostrado anteriormente (Figura 3).

Periodicamente, todos os dados coletados são analisados e decisões são tomadas em quais áreas dentro da plataforma e do ensino geral alterações devem ocorrer para melhorar o aprendizado. As informações disponibilizadas permitem o professor avaliar o aproveitamento do curso e as necessidades individuais e coletivas dos alunos (Figura 3).

Figura 1 – Ambiente de Desenvolvimento Integrado.

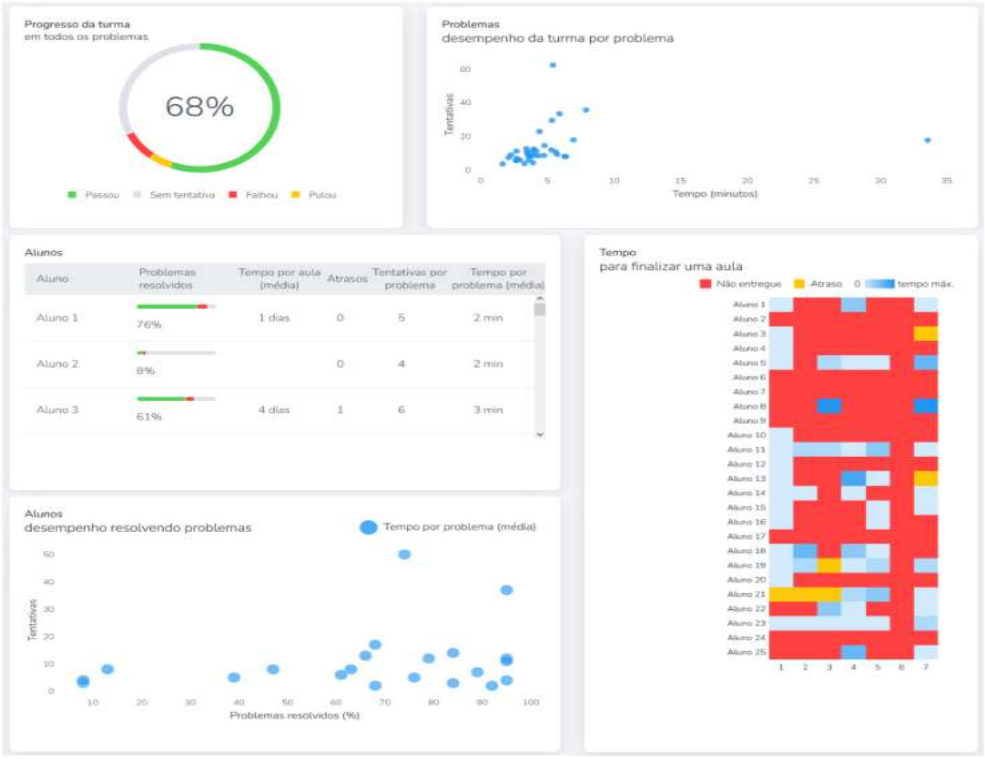


Fonte: Machine Teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação (MORAES et al., 2022, p. 5).

Figura 2 – Painel com estatísticas do aluno.

Fonte: Machine Teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação (MORAES et al., 2022, p. 6).

Figura 3 – Painel para uma turma no Machine Teaching.



Fonte: Machine Teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação (2022, p. 8).

Dada a importância dos dados coletados, é crucial que os testes sejam bem elaborados para que a precisão das informações retiradas sejam correspondentes às necessidades dos alunos e, com isso, medidas competentes com relação ao ensino sejam feitas. Adicionalmente, testes que deixem passar códigos com defeitos levarão o aluno a ter a falsa impressão de estar conseguindo resolver os problemas quando na realidade não está. Como já existem métodos de projeto de testes utilizados na área da computação, implementá-los nas criações dos testes desses exercícios deve ajudar na elaboração de testes melhores.

2.2 TESTES E CRITÉRIOS

Testes são execuções que tentam identificar defeitos (assumindo que defeitos devem estar presentes). Eles tentam descobrir se dados certos parâmetros que correspondem a certos requisitos, o código apresenta um comportamento esperado, ou seja, o resultado previsto nos testes é o mesmo do resultado apresentado. Se eles não forem iguais, isso demonstra que o **teste falhou** e, consequentemente, a existência de um **defeito** no código. Defeitos são possíveis problemas contidos no código que podem ou não alterar o resultado final. Já falhas são as manifestações de comportamentos incorretos que vão contrário ao propósito e requisitos do programa. Assim sendo, falhas indicam defeitos no código (AMMAN; OFFUTT, 2017, seção 1.1). Os conjuntos de testes mais interessantes são

aqueles que conseguem apontar a maior quantidade de falhas possível. Com isso, testes aumentam o nível de confiança de que o software está alcançando seu objetivo com a ressalva de que não é possível fazer teste exaustivo.

Uma maneira de direcionarmos a criação do projeto de teste é a utilização de critérios de cobertura. A criação de testes depende de seus requisitos e do critério de cobertura escolhido. Os requisitos de teste são elementos específicos de um artefato de software que um caso de teste deve satisfazer ou cobrir. Já um critério de cobertura é uma coleção de regras que definem como os requisitos de testes devem ser criados (AMMAN; OFFUTT, 2017, cap. 6). Tais critérios permitem diminuir redundância nos testes e o tempo necessário para executá-los. Neste projeto, todos os testes foram avaliados em um tipo de critério de cobertura específico: **particionamento de entrada**.

Nesse critério, dado a entrada do programa, as características daquela entrada são identificadas e cria-se um particionamento para ela. Tal particionamento é composto por blocos de forma que um valor específico não possa estar em dois blocos pertencentes ao mesmo domínio. Dessa forma, assumindo um domínio hipotético D separado em 3 blocos, b_1, b_2 e b_3, temos que os valores pertencentes a b_1 não estão contidos em b_2 e b_3, valores de b_2 não estão contidos em b_1 e b_3, e valores de b_3 não estão contidos em b_1 e b_2 (AMMAN; OFFUTT, 2017, cap. 6).

Esse particionamento é definido utilizando características relevantes do programa como os tipos dos parâmetros de entrada, o intervalo dos valores (caso sejam valores ordenáveis), relevância contextual (tal valor pode ocasionar um comportamento errôneo no programa, por exemplo), ou qualquer outro fator que a pessoa criando os testes ache relevante ser avaliado. Assim, o critério fornece heurísticas que dão direções de como explorar o espaço de entrada sem fazer teste exaustivo. Feito isso, os blocos são criados e assume-se que todos os valores de um bloco devem se comportar da mesma maneira ao serem testados.

Para exemplificar esse processo, vamos assumir que temos um programa cujo objetivo é receber uma letra e uma frase qualquer e retornar a última posição da letra na frase se ela existir ou -1 se não existir, e devemos criar testes para esse programa. Assumiremos que as características relevantes dos testes podem ser:

a) Se a letra é maiúscula ou minúscula (let):

    1. A letra é maiúscula (let_maius).
    2. A letra é minúscula (let_minus).

b) Quantas ocorrências da letra acontecem na frase (let_oco):

    1. A letra não está na frase (let_oco_i0).
    2. A letra está somente uma vez na frase (let_oco_i1).
    3. A letra está múltiplas vezes na frase (let_oco_M1).

c) Quantidade de palavras na frase (frase_pal):

1. A frase tem só uma palavra (frase_pal_i1).
2. A frase tem mais que uma palavra (frase_pal_M1).

Como dito anteriormente, em um mesmo particionamento, nenhuma entrada pode pertencer a mais de um bloco e todas as entradas possíveis encaixam em exatamente um bloco de cada particionamento. Na documentação, cada bloco é identificado com uma abreviatura que fornece uma intuição de sua característica principal. Por exemplo, o bloco let_oco_i0 contém todas as entradas onde a ocorrência da letra na frase é igual a zero.

Uma vez feito isso, 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 (AMMAN; OFFUTT, 2017, p.132). Quanto mais amplo é o critério de combinação, mais preciso será captar todos os tipos de comportamentos e, ao mesmo tempo, mais tempo levará para executar todos os testes. Nesse projeto aplicou-se somente três tipos de combinações:

- All Combinations Coverage: Todas as combinações possíveis de todos os blocos, onde toda combinação deve incluir um bloco de cada particionamento.
- Pairwise Coverage: Todas as combinações possíveis em pares de blocos de particionamentos diferentes.
- Each Choice Coverage: Um valor de cada bloco de 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:
  (let_maius, let_oco_i0, frase_pal_i1), (let_maius, let_oco_i0, frase_pal_M1),
  (let_maius, let_oco_i1, frase_pal_i1), (let_maius, let_oco_i1, frase_pal_M1),
  (let_maius, let_oco_M1, frase_pal_i1), (let_maius, let_oco_M1, frase_pal_M1),
  (let_minus, let_oco_i0, frase_pal_i1), (let_minus, let_oco_i0, frase_pal_M1),
  (let_minus, let_oco_i1, frase_pal_i1), (let_minus, let_oco_i1, frase_pal_M1),
  (let_minus, let_oco_M1, frase_pal_i1), (let_minus, let_oco_M1, frase_pal_M1)
- PairWise Coverage:
  (let_maius, let_oco_i0), (let_maius, let_oco_i1), (let_maius, let_oco_M1),
  (let_minus, let_oco_i0), (let_minus, let_oco_i1), (let_minus, let_oco_M1),
  (let_maius, frase_pal_i1), (let_maius, frase_pal_M1),
  (let_minus, frase_pal_i1), (let_minus, frase_pal_M1),

(let_oco_i0, frase_pal_i1), (let_oco_i0, frase_pal_M1),
(let_oco_i1, frase_pal_i1), (let_oco_i1, frase_pal_M1),
(let_oco_M1, frase_pal_i1), (let_oco_M1, frase_pal_M1)

- Each Choice Coverage: (let_maius, let_minus, let_oco_i0, let_oco_i1, let_oco_M1, frase_pal_i1, frase_pal_M1)

Apesar das combinações serem feitas para todos os programas, não listamos elas dessa forma nesse trabalho. Utilizamos uma tabela para realizar as combinações pois oferece uma maior praticidade na hora de criar os testes e é mais agradável visualmente. Essa tabela é demonstrada e explicada no capítulo 3 (Figura 4). Terminada essa etapa, são elaborados os casos de testes que atendem a esses requisitos. Vamos utilizar a cobertura Pairwise para criar o conjunto de testes:

- Caso de teste com entrada (‘A’, ‘arvore’) satisfaz (let_maius, let_oco_i0), (let_maius, frase_pal_i1), e (let_oco_i0, frase_pal_i1).
- Caso de teste com entrada (‘A’, ‘Arvore baixa’) satisfaz (let_maius, let_oco_i1), (let_maius, frase_pal_M1), e (let_oco_i1, frase_pal_M1).
- Caso de teste com entrada (‘A’, ‘A Amanda é baixa’) satisfaz (let_maius, let_oco_M1), (let_maius, frase_pal_M1), e (let_oco_M1, frase_pal_M1).
- Caso de teste com entrada (‘a’, ‘bombom gostoso’) satisfaz (let_minus, let_oco_i0), (let_minus, frase_pal_M1), e (let_oco_i0, frase_pal_M1).
- Caso de teste com entrada (‘a’, ‘boia’) satisfaz (let_minus, let_oco_i1), (let_minus, frase_pal_i1), e (let_oco_i1, frase_pal_i1).
- Caso de teste com entrada (‘a’, ‘barbado’) satisfaz (let_minus, let_oco_M1), (let_minus, frase_pal_i1), e (let_oco_M1, frase_pal_i1).

A criação dos testes de forma concreta varia de acordo com a linguagem de programação, assim como a execução dos mesmos, mas os requisitos e os blocos criados podem ser utilizados independentemente da linguagem.

Um dos principais benefícios do particionamento de entrada é na depuração do código. Como os testes são projetados em cima das especificações do projeto e não do gabarito, isso permite uma facilidade na hora de averiguar qual defeito que um código, que falhou em um teste específico, contém dados os requisitos que aquele teste está satisfazendo. Um exemplo baseado nos requisitos acima seria se o código falhar em testes em que a letra não está na frase, isso mostra que o código não trata esse requisito específico. Isso também demonstra que a qualidade do particionamento criado muito influencia a qualidade dos testes, já que particionamentos que não dividem bem o espaço de entrada não conseguirão captar uma grande quantidade de defeitos.

2.3 EXPERIMENTO ORIGINAL DE MODELAGEM DE TESTES DE SOFTWARE PARA MACHINE TEACHING

Como dito anteriormente, o projeto do grupo anterior, a partir do qual desenvolvemos o nosso, recolheu uma amostra dos problemas, códigos e resultados relacionados aos testes cadastrados na plataforma para averiguar a diferença de desempenho entre os testes cadastrados na plataforma e os criados utilizando critérios rigorosos na modelagem de testes (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023). Utilizaram três tipos de critérios de cobertura (AMMAN; OFFUTT, 2017): particionamento de entrada, grafo e mutação, executando os códigos recebidos pela plataforma utilizando os testes projetados pelo grupo.

Para realizar o experimento, eles analisaram 33 problemas cadastrados na plataforma, recebendo os códigos submetidos pelos alunos nos respectivos exercícios. Depois, criaram requisitos e casos de testes, de acordo com os critérios e coberturas selecionadas. Todos os dados estão no arquivo pickles.zip, o qual contém arquivos no formato Pickle: um arquivo problems.pkl contendo os enunciados dos problemas e um arquivo solutions.pkl contendo todas as submissões de códigos feitas por alunos em conjunto com um valor que representa se o código passou em todos os testes (valor 1) ou se falhou em pelo menos um teste (valor 0). O grupo utilizou esse valor para comparação entre os resultados dos testes deles e os da Machine Teaching.

O experimento realizado pelo grupo foi estruturado utilizando scripts em Python cuja principal parte da execução é feita por dois arquivos: runner.py e save_tests_output_main.py. A estrutura da execução funciona da seguinte forma:

1. Dentro do programa *runner.py* é registrado o número dos exercícios que são executados e os critérios de testes e é feita uma combinação desses, criando uma lista de listas com cada elemento sendo uma lista composta pelo número de um dos problemas e o nome do critério correspondente ao conjunto de teste que quer ser analisado.

2. Após isso, o *runner.py* executa uma função da biblioteca Joblib [1] chamada Parallel, essa função chama a função *save_tests_output* do programa *save_tests_output_main.py*, em conjunto com os parâmetros dela que são dados por cada elemento da lista anterior. Essa função cria um processo para cada chamada da função *save_tests_output*, "iterando"sobre a lista de combinações de forma paralelizada.

3. Dentro do *save_tests_output*, o programa cria um novo laço onde cada iteração executa os testes elaborados pelo grupo em conjunto com cada um dos códigos dos alunos submetidos àquele problema na plataforma, utilizando a bilioteca PyTest para isso.

[1] https://joblib.readthedocs.io/en/stable/

4. Ao fim da execução de todos os códigos, registra-se os resultados (quais testes falharam, passaram, deram erro ou passaram do tempo máximo) dentro de duas planilhas: uma tem o resultado de cada execução de cada solução e outra tem um resumo dos resultados para cada caso de teste individual, utilizando a biblioteca Pandas para essa etapa.

Após isso, utilizou-se o arquivo *aggregate_ data.ipynb* para criar gráficos para análise dos dados recebidos. Foram criados: um mapa de calor, um gráfico de pizza e um gráfico de barra.

A métrica utilizada para comparar os desempenhos entre os diferentes conjuntos de testes foi a quantidade de "defeitos inéditos", de criação do próprio grupo, como descrito na página 36:

> 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 é reprovada por 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.

Com essa métrica, foram criados três cenários para classificar os códigos dos alunos (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023, p. 37):

- Caso de concordância: Quando todos os 4 resultados agregados são iguais.
- 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.
- 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. Chamaram essa categoria de defeitos inéditos.

Foi concluído então que os testes novos aumentaram a identificação dos defeitos inéditos em aproximadamente 4.2% juntando os três critérios. Contudo foi declarado que esse número pode ter sido distorcido devido a plataforma Machine Teaching corrigir alguns defeitos antes do teste devido ao método de implementação da ferramenta, como descrito no texto do trabalho, na página 35:

> 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.

3 METODOLOGIA

A proposta desse projeto é uma nova análise do projeto "MODELAGEM DE TESTES DE SOFTWARE - Uma Análise dos Resultados de Testes em Exercícios de Programação" voltada para o critério de análise de particionamento de entrada em conjunto com a averiguação dos erros mais comuns encontrados nos testes.

O projeto foi realizado da seguinte forma:

1. Realizar um novo particionamento de entrada e criação de testes de todos os exercícios originalmente analisados pelo grupo do projeto acima mencionado. Como o desempenho dos testes é altamente influenciado pelo particionamento feito, faz sentido executar essa etapa novamente (Subseção 3.1).

2. Obter submissões de códigos dos alunos, código do projeto do grupo anterior, e testes da Machine Teaching. Esses as submissões estão disponíveis num repositório remoto[1] no formato *.pkl*, os códigos do projeto do grupo na plataforma GitHub[2], e os testes da Machine Teaching foram obtidos manualmente na própria plataforma (Subseção 3.2).

3. Modificação dos códigos que efetuavam a execução dos testes e registro dos resultados, esses obtidos na etapa anterior, com o objetivo de considerar somente o critério de particionamento de entrada (Subseção 3.3). O arquivo com os scripts que criam gráficos também foi alterado.

4. Execução de todos os códigos dos alunos em conjunto com os testes (Subseção 3.4).

5. Averiguação das porcentagens de falhas dos testes da plataforma Machine Teaching e desse projeto e detalhamento dos erros mais comuns encontrados entre todos os testes, e compilação dos achados em guias para criação de enunciados e projeto de testes. (Subseção 3.5)

Os exercícios e soluções foram obtidos pelo grupo anterior com o apoio da equipe de desenvolvimento da plataforma Machine Teaching e disponibilizados para esse projeto. Os arquivos não contém informações pessoais sobre os autores, mantendo o anonimato dos alunos. Todos os dados estão no arquivo pickles.zip, o qual contém arquivos no formato Pickle: um arquivo problems.pkl contendo os enunciados dos problemas e um arquivo solutions.pkl contendo todas as submissões de código enviadas por alunos.

[1] https://drive.google.com/file/d/1CFEO6PHUJf5DDRLEZen9vCJ-_X97yM_Z/view?usp=sharing
[2] https://github.com/troclaux/tcc-machine-teaching

3.1 CRIAÇÃO DE NOVOS PARTICIONAMENTOS E CASOS DE TESTES

A criação dos novos particionamentos foi feita baseada na interpretação dos enunciados dos problemas levando em considerações características críticas mencionadas anteriormente, os tipos de entrada que os problemas recebem, e os particionamentos criados pelo grupo anterior. Foram criadas planilhas para cada exercício com a descrição dos particionamentos e seus respectivos blocos em conjunto com uma tabela separada para a criação dos casos de teste. Um link para a pasta com as tabelas está disponibilizado no apêndice desse arquivo.

Figura 4 – Exemplo de Tabela de Criação de Testes para Cobertura Pairwise.

	frase_simb_inicio_S	frase_simb_inicio_N	frase_simb_fim_S	frase_simb_fim_N		
frase_simb_fim_S	CT4	CT3			CT1:	('cachorro', 1)
frase_simb_fim_N	CT2	CT1			CT2:	(' gatinho', 1)
frase_qtde_i0	CT6	CT5	CT6	CT5	CT3:	('macio ', 1)
frase_qtde_i1	CT2/4	CT1/3	CT3/4	CT1/2	CT4:	(' 15mil ', 1)
frase_qtde_M1	CT8	CT7	CT8	CT7	CT5:	('', 0)
					CT6:	(' ', 0)
					CT7:	('para quedas', 2)
					CT8:	(' valeu amigo ', 2)

Na Figura 4, pode-se observar os blocos correspondentes a cada particionamento posicionados de forma a criar combinações dois a dois de blocos de particionamentos diferentes, com casos de testes cumprindo os requisitos estabelecidos por esse par de blocos. Ao lado, cada teste está representado por uma tupla que contém as entradas do programa seguidas pelo retorno esperado. As células em preto são simplesmente combinações não existentes. Elas tem uma cor de fundo pois não contém conteúdo algum. Já cores pretas representam combinações inválidas de blocos, as quais não existem casos de testes para supri-las. Com foco de diminuir a quantidade de casos de testes, se um determinado problema tivesse uma quantidade muito alta de particionamentos seria escolhido para esse problema o critério de combinação Each Choice, do contrário sempre foi utilizado Pairwise, que em certas ocasiões equivalia ao All Combinations. Existem dois motivos para isso: grande parte dos problemas recebe somente entre dois ou três parâmetros, e, baseado em dados empíricos, a maior parte das falhas são observadas por meio de um parâmetro ou uma combinação de dois a seis parâmetros (KUHN et al., 2015). Assim, nesse projeto, o Pairwise foi determinado como um bom meio-termo entre a quantidade de testes e seus prováveis desempenhos.

3.2 OBTENÇÃO DOS CÓDIGOS, PROJETO E TESTES

Os códigos do projeto MODELAGEM DE TESTE DE SOFTWARE e as submissões utilizadas foram disponibilizadas pelo grupo anterior em repositórios online. Contudo, para esse projeto, foi decidido coletar os testes no site do Machine Teaching ao invés de

utilizar os resultados que já estavam armazenados dentro das planilhas. Tal decisão foi tomada pois os resultados colhidos anteriormente são relacionados a testes mais desatualizados e podem acarretar em uma defasagem na proporção de falhas em relação aos testes mais novos. Assim, os testes mais atualizados (até abril de 2024) foram adicionados aos critérios de execução.

3.3 MODIFICAÇÃO DO CÓDIGO

Algumas modificações foram feitas pois havia funcionalidades que não eram necessárias para esse projeto e outras que precisavam ser adaptadas para os testes atuais.

1. Foram removidas todas as instâncias de utilização dos critérios de mutação e grafo e foi adicionado um novo critério chamado *new_input* correspondente aos novos casos de testes de particionamento de entrada.

2. Foram removidos os resultados dos testes originais que já estavam cadastrados e foi criado um critério chamado *original* correspondente aos testes adquiridos na plataforma Machine Teaching. Isso foi feito para se obter um resultado mais atualizado.

3. O arquivo *save_tests_output_main.py* foi reformulado para ter uma função main que chama a função *save_tests_output*. Isso foi feito para que certos passos, como pegar os caminhos das planilhas de armazenamento, fossem executados só uma vez.

4. Foi descartado o uso da biblioteca Joblib e utilizado os módulos nativos do Python para empregar a paralelização.

5. Foi desconsiderada a utilização da planilha que armazena o resultado acumulado de cada caso de teste específico. Ela não é necessária pois é possível adquirir essas informações após a execução dos testes, diminuindo o tempo total de execução por necessitar de menos escrita.

6. O arquivo *aggregate_data.ipynb* foi alterado pois os gráficos não foram utilizados nesse projeto. Foram criadas duas funções: uma averígua as porcentagens nos cenários de discordância e concordância de resultados e outra cria um gráfico de caixa somente nos cenários de discordância ("FP", onde os códigos dos alunos falharam nos testes da Machine Teaching e passaram nos testes do projeto atual, e "PF", onde os códigos dos alunos passaram nos testes da Machine Teaching e falharam nos testes do projeto atual).

O grupo anterior usou uma biblioteca chamada Joblib dentro do *runner.py* para implementar a paralelização. Já nesse projeto é utilizado o módulo multiprocessing do Python para isso. Basicamente, um laço chama a função main em *save_tests_output_main.py*

tendo como parâmetros o número do problema e o nome do critério. Dentro de *main*, é feito etapas de pré-processamento, como pegar os caminhos das planilhas para armazenar os resultados, pegar os números de identificação dos códigos dos alunos, etc. Ali, também é chamada a função *chunk_gen* cujo o propósito é dividir a lista contendo os números de identificação dos códigos dos alunos em pequenos grupos de tamanhos aproximados. Esses grupos serão entregues, cada um, a um processo para que os códigos dos alunos correspondentes aos números sejam testados. Essa paralelização foi feita usando a função *pool.imap_unordered*, que chama a função *save_tests_output* em conjunto com o subgrupo com os números de identificação dos alunos, os caminhos para as planilhas, códigos, etc, para cada processo. Uma vez que o processo termine, os dados são armazenados nas planilhas correspondentes ao número do problema.

Essa alteração foi feita pois o grupo anterior paralelizava a nível de problema, onde cada processo executava todas as submissões de código dos alunos para um problema específico e os testes de um critério específico, de forma que se for necessário executar menos combinações de problema e critério do que a quantidade de processos que o computador pode criar em paralelo, haveriam núcleos ociosos, tornando o processo mais lento do que necessário. Como nesse projeto havia necessidade de por vezes executar um único problema e critério em específico, foi escolhido trabalhar de forma sequencial ao nível de problema, mas paralelizar os códigos dos alunos, dividindo o conjunto de códigos dos alunos em pequenos grupos e usando um processo para cada um desses subconjuntos.

3.4 EXECUÇÃO DOS TESTES E REGISTRO DOS RESULTADOS

Os testes foram executados duas vezes: uma vez no laboratório LC3 (Laboratório de Combinatória e Computação Científica) na Ilha do Fundão, no Rio de Janeiro e outro em um computador doméstico. Na execução no computador do LC3 foram utilizados seis núcleos do processador e demorou cerca de dezoito horas para executar os códigos com os testes do new_input e o input (correspondente ao projeto anterior). Já no computador doméstico foram executados o códigos com os testes do original e new_input (algumas reexecuções foram necessárias). Foi um utilizado um processador de dez núcleos e demorou cerca de nove horas (algumas das mudanças no código foram feitas após a primeira execução, as quais ajudaram a acelerar o processo).

Os testes foram aplicados usando a biblioteca PyTest, em conjunto com alguns plugins para facilitar a automatização. O programa salva o resultado como strings dentro de uma planilha que contém as seguintes colunas:

- *solution_id*: número de identificação único de cada submissão dos alunos.
- *problem_id*: número do problema.

- *outcome*: uma coluna para cada critério de teste, correspondente a um resultado, *True*, se passou em todos os testes, ou *False*, se falhou em pelo menos um dos testes daquele critério.

- *result*: string que é a sequencia de resultados dos testes daquela submissão de código (*solution_id*). Essas strings tem todas um tamanho igual à quantidade de testes que o critério correspondente tem, com a posição da letra sendo corresponde à posição do teste dentro da lista de testes em seus correspondentes arquivos. Os resultados possíveis representados pelas letras são:

  – "P": Passou. O resultado entregue pelo código foi igual ao resultado esperado.

  – "F": Falhou. O resultado entregue pelo código foi diferente do resultado esperado.

  – "E": Erro. O código encontrou um erro durante a execução.

  – "T": Timeout. O código passou do tempo máximo disponível para retornar um valor (o valor máximo padrão escolhido foi de um segundo).

  Ou seja, por exemplo, na string "PFET", a primeira letra representa o resultado do primeiro teste executado, o qual foi aprovado; a segunda, do segundo, o qual foi reprovado; a terceira representa do terceiro teste, o qual deu erro durante a execução; e a quarta representa o resultado do quarto teste, que passou do tempo máximo de execução permitido.

Um recorte do início da planilha correspondente ao problema 800 pode ser vista na Figura 5.

Figura 5 – Exemplo de planilha com os resultados agregados de um problema.

solution_id	problem_id	original_outcome	input_outcome	new_input_outcome	original_result	input_result	new_input_result
340229	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340230	800	False	False	False	EEEEEEEEEE	EE	PEPPEEE
340231	800	False	False	False	PPPFPPPFPP	FP	PPPPPFF
340232	800	False	False	False	PPPPPPFFPP	FP	PEPEPEP
340233	800	False	False	False	PPPPPPFFPP	FP	PEPEPEP
340234	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340235	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340236	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340237	800	False	False	False	FFFFFFFFFF	FP	PEPEPEF
340238	800	False	False	False	PPPPPPFFPP	FP	PEPEPEP
340239	800	False	False	False	PPPFPPPFPP	FP	PPPPPFF
340240	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340241	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340242	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340243	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340244	800	False	False	False	EEEEEEEEEE	EE	EEEEEEE
340245	800	False	False	True	PPPPPPFFPP	FP	PPPPPPP
340246	800	False	False	True	PPPPPPFFPP	FP	PPPPPPP
340247	800	False	False	True	PPPPPPFFPP	FP	PPPPPPP

3.5 AVERIGUAÇÃO DAS TAXAS DE FALHAS E DEFEITOS

Para comparar desempenhos, foram contados os resultados dos testes da Machine Teaching e desse projeto em todos os problemas, dividindo esse valor em porcentagens em diferentes cenários de discrepância ou concordância da coluna *outcome*. Os resultados da execução dos testes do grupo anterior não foram utilizados nessa comparação. Isso foi feito dentro do arquivo *aggregate_ data.ipynb*, utilizando as planilhas *solution* de cada problema. Os cenários de concordância e discrepância utilizados estão dispostos na lista abaixo.

- "FF": nesse cenário, o valor na coluna *outcome* e na linha correspondente ao código do aluno está em *False*, tanto no critério *original* quanto no critério *new_ input*.
- "FP": nesse cenário, o valor na coluna *outcome* e na linha correspondente ao código do aluno está está em *False* no critério *original*, e está em *True* no critério *new_ input*.
- "PF": nesse cenário, o valor na coluna *outcome* e na linha correspondente ao código do aluno está está em *True* no critério *original*, e está em *False* no critério *new_ input*.
- "PP": nesse cenário, o valor na coluna *outcome* e na linha correspondente ao código do aluno está em *True*, tanto no critério *original* quanto no critério *new_ input*.

Já para verificar os defeitos mais comuns entre as submissões dos alunos, foi escolhido comparar os resultados dos testes da Machine Teaching, do grupo anterior, e desse projeto. Isso foi feito como forma de facilitar um "cruzamento de informações" entre os testes que falharam e passaram em cada cenário, permitindo estipular quais possíveis requisitos deixaram de ser cobertos. Os cenários descritos então ficaram como escritos abaixo.

- "FFF": todos os testes falharam.
- "PFF": os testes originais passaram e pelo menos um dos testes do projeto anterior e desse projeto falharam.
- "FPF": os testes do projeto anterior passaram e pelo menos um dos testes originais e desse projeto passaram.
- "FFP": os testes desse projeto passaram e pelo menos um dos testes originais e do projeto anterior falharam.
- "FPP": os testes do projeto anterior e desse projeto passaram e pelo menos um dos testes originais falharam.

- "PFP": os testes originais e desse projeto passaram e pelo menos um dos testes do projeto anterior falharam.
- "PPF": os testes originais e do projeto anterior passaram e pelo menos um dos testes desse projeto falharam.
- "PPP": todos os testes passaram.

Uma vez determinada as porcentagens, cada um dos cenários foi analisado individualmente utilizando as planilhas com os resultados agregados dos testes em conjunto com filtros, olhando cada cenário individualmente, como demonstrado na Figura 6.

Figura 6 – Utilizando filtros na tabela 800 para isolar códigos no cenário PFF.

solution_	problem_	original_outcom	input_outcom	new_input_outcom	original_result	input_resu	new_input_resu
340343	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340358	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340503	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340519	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340520	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340521	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
340522	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341226	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341228	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341377	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341420	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341744	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341841	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
341870	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF
342078	800	True	False	False	PPPPPPPPPP	FP	PPPPPFF

A partir daí, alguns códigos eram copiados em um pequeno script, feito para olhar os resultados do código do aluno de uma forma mais detalhada, em conjunto com os testes. A partir da execução desse script e baseado nos resultados discrepantes entre os testes e o fornecido pelos códigos, foi estimado quais requisitos possíveis deixaram de ser cobertos. Quando a quantidade de códigos era muito grande para ser analisado individualmente, uma amostra dos códigos era pega, pois muitos acabavam apresentando os mesmo defeitos. Esses erros comuns foram usados para o guia de criação de enunciado e projeto de testes mas a planilha onde estão registrados os achados de forma resumida está disponível em um repositório remoto [3].

3.6 MÉTRICAS

Antes de falar da análise, deve-se discutir um pouco sobre quais as métricas usadas e como elas foram escolhidas. Como comentado anteriormente, o grupo anterior utilizou o conceito de "defeito inédito" para medir o desempenho entre os conjuntos de testes, um conceito que também foi utilizado nesse trabalho. Então, o desempenho de cada critério

[3] https://docs.google.com/spreadsheets/d/1EjmRO7cK5ZQN4JMpMpfo17Wuo2F1oAGQCcn32m1ke6U/edit?usp=sharing

foi estimado baseado na quantidade de submissões defeituosas que foram identificadas durante a execução.

A eficácia dos testes novos será medida a partir de falhas que não foram identificadas pelo conjunto de testes cadastrados na plataforma Machine Teaching. A lógica para usar essa métrica está na seguinte linha de raciocínio: se uma solução passa os testes originais, mas falha em qualquer teste novo, esta solução provavelmente apresenta um defeito que não foi detectado originalmente.

Os resultados dos testes de um critério específico foram agregados em um único valor. Dessa forma, em um critério qualquer, se um ou mais casos de teste resultarem em erro de execução ou falha, o resultado agregado desse critério falha (nomenclatura "F"), do contrário o resultado agregado passa (nomenclatura "P"). No caso desse trabalho existiram três tipos: um resultado agregado para os testes originais, um para os testes do grupo anterior, e um para o feito nesses trabalho.

Assim, as métricas utilizadas foram baseadas nas seguintes suposições: seja p um problema pertencente ao conjunto de problemas $Prob$. Seja cada submissão s_p pertencente ao total de submissões S_p para um problema p. Cada conjunto de testes corresponde a um critério específico e quando executado por uma submissão s_p retorna um valor do conjunto {0, 1} com 1 representando que a submissão passou em todos os testes e 0, que falhou em pelo menos um teste daquele conjunto. Assim, seja o resultado das execuções da submissão s_p com o conjunto de testes da Machine Teaching $MT(s_p) = \{1, 0\}$ e para os dos testes desse projeto $AT(s_p) = \{1, 0\}$, para os casos acima especificados.

Assim, dado um problema p, declaramos:

- $Fail_p$ o conjunto de todas as submissões s_p em S_p tal que $MT(s_p) = AT(s_p) = 0$, ou seja, submissões identificadas como defeituosas por ambos os conjuntos de testes.
- $Pass_p$ o conjunto de todas as submissões s_p em S_p tal que $MT(s_p) = AT(s_p) = 1$, ou seja, submissões aceitas por ambos os conjuntos de testes.
- AT_p o conjunto de todas as submissões s_p em S_p tal que $MT(s_p) = 1$ e $AT(s_p) = 0$, ou seja, submissões que apenas o conjunto de testes novo identificou como defeituosas.
- MT_p o conjunto de todas as submissões s_p em S_p tal que $MT(s_p) = 0$ e $AT(s_p) = 1$, ou seja, submissões que apenas o conjunto de testes da Machine Teaching identificou como defeituosas.

Então, se para um problema p, $|AT_p| > |MT_p|$[4], o conjunto de testes AT teve melhor desempenho, ou seja, identificou uma maior quantidade de submissões defeituosas, e vice-versa, se $|MT_p| > |AT_p|$, o conjunto de testes originais da Machine Teaching apresentou melhor desempenho.

[4] $|A|$ representa a cardinalidade do conjunto A.

Para a avaliação geral, considerando o conjunto de problemas $Prob$ as estatísticas escolhidas foram média, mediana e total de problemas com melhor desempenho de um ou outro conjunto de testes, calculadas das seguintes formas:

- Média :
  - Cenário "Ambos Falharam": $(1/|Prob|) * \sum_{p \in Prob} |Fail_p|/|S_p|$.
  - Cenário "MT falhou e atual passou": $(1/|Prob|) * \sum_{p \in Prob} |MT_p|/|S_p|$.
  - Cenário "MT passou e atual falhou": $(1/|Prob|) * \sum_{p \in Prob} |AT_p|/|S_p|$.
  - Cenário "Ambos Passaram": $(1/|Prob|) * \sum_{p \in Prob} |Pass_p|/|S_p|$.
- Mediana (assuma que um conjunto A contém todos valores $|A_p|/|S_p|$ para todo p em $Prob$, e assuma que A está ordenado):
  - Cenário "Ambos Falharam": elemento $Fail_{(|Prob|+1)/2}$.
  - Cenário "MT falhou e atual passou": elemento $MT_{(|Prob|+1)/2}$.
  - Cenário "MT passou e atual falhou": elemento $AT_{(|Prob|+1)/2}$.
  - Cenário "Ambos Passaram": elemento $Pass_{(|Prob|+1)/2}$.
- Total de problemas com porcentagem superior em cenários discordantes:
  - Cenário "MT falhou e atual passou": $|\{\forall p \in Prob | \forall MT_p \in MT, \forall AT_p \in AT, MT_p > AT_p\}|$.
  - Cenário "MT passou e atual falhou": $|\{\forall p \in Prob | \forall MT_p \in MT, \forall AT_p \in AT, AT_p > MT_p\}|$.

O resultado agregado está organizado nos seguintes cenários: códigos que falharam nos testes cadastrados na Machine Teaching (referida como MT) e nos testes atuais criados por esse projeto, códigos que falharam nos testes da Machine Teaching mas passaram nos desse projeto, códigos que passaram nos testes da Machine Teaching mas falharam nos desse projeto, e códigos que passaram em ambos. Os casos de testes do grupo anterior estão sendo desconsiderados inicialmente para que a comparação seja objetiva. Já para averiguar os erros mais comuns, foram utilizados os testes originais, os testes do grupo anterior (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023), e os testes desse projeto, como descrito anteriormente, de forma a ter maior facilidade em entender quais requisitos estão sendo atendidos em cada cenário.

4 RESULTADOS E ANÁLISES DOS EXPERIMENTOS

O primeiro objetivo dessa análise é atestar se a metodologia de projeto de teste por particionamento de entrada é uma boa opção para o projeto de testes para correção automática em ferramentas de introdução à programação. Como pode se observar na tabela 1, em 27 dos 35 exercícios, houve uma porcentagem maior de falhas nos casos de testes novos do que nos originais, com uma média de 1.00% para falhas nos casos originais e aprovações nos novos testes, e 3.82% para falhas nos casos novos e aprovações nos originais. Devido a termos alguns casos onde os números ficaram muito grandes, foi criado um gráfico de caixa (Figura 7) de ambos os cenários para averiguar a mediana e a distribuição geral dos dados, já que a mediana indica o ponto médio da amostra e é pouco afetado por valores extremamente grandes (COX; JONES, 1981, p. 138). A mediana ficou com um valor aproximado de 0.16% para o cenário nos quais os códigos falharam nos testes da Machine Teaching e passaram nos desse projeto e 1.31% para o cenário em que os códigos passaram nos testes da Machine Teaching e falharam nos desse projeto. As quantidades de códigos e os resultados percentuais podem ser observados na tabela 1. Já a quantidade de testes e os resultados relativos aos cenários em discordância podem ser observados na tabela 2.

Figura 7 – Gráfico de caixa dos cenários de discordância.

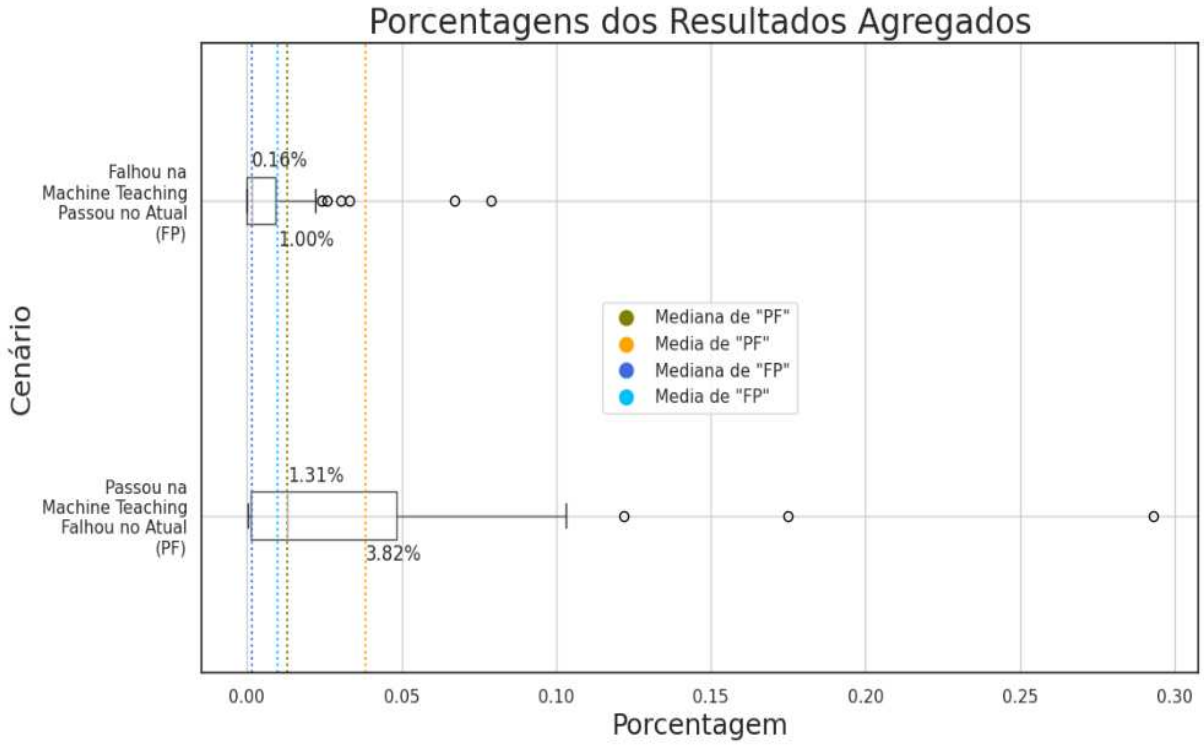


Tabela 1 – Resultado agregado absoluto.

Nº do problema	Quantidade de códigos	Ambos falharam	MT falhou e e atual passou	MT passou e atual falhou	Ambos passaram
736	6706	62.50%	0.00%	0.03%	37.47%
742	16956	84.85%	1.37%	0.01%	13.77%
744	10403	78.95%	0.07%	0.10%	20.89%
751	6721	69.36%	0.00%	29.30%	1.34%
798	5973	80.90%	2.19%	4.14%	12.77%
800	4738	72.84%	6.71%	8.44%	12.01%
804	26687	92.76%	0.39%	0.12%	6.73%
806	13821	88.47%	3.33%	1.22%	6.98%
807	21940	90.58%	1.15%	1.75%	6.52%
809	10092	81.36%	0.00%	0.12%	18.52%
810	20115	91.52%	0.00%	1.94%	6.54%
811	10400	86.74%	0.70%	7.35%	5.21%
812	13761	86.10%	0.00%	4.49%	9.41%
815	8689	79.27%	0.00%	0.13%	20.60%
816	12510	82.12%	3.05%	9.94%	4.89%
817	14486	87.95%	0.33%	1.48%	10.24%
819	11342	85.56%	0.00%	0.36%	14.08%
820	13620	84.39%	0.02%	10.32%	5.27%
821	6730	75.36%	0.00%	5.20%	19.44%
822	7920	81.40%	0.38%	5.90%	12.32%
823	10395	86.63%	0.65%	2.76%	9.96%
824	15247	81.77%	7.88%	0.37%	9.97%
827	8019	78.01%	2.43%	0.01%	19.54%
828	8491	79.80%	0.32%	17.48%	2.40%
829	5414	67.92%	0.46%	3.97%	27.65%
831	9174	83.45%	0.20%	12.19%	4.16%
832	10474	85.10%	0.69%	0.07%	14.15%
833	7580	81.45%	0.00%	0.13%	18.42%
834	5650	74.30%	0.00%	0.39%	25.31%
835	10078	85.54%	0.05%	1.31%	13.10%
836	7277	86.93%	0.54%	1.92%	10.61%
838	7151	70.63%	0.06%	0.25%	29.06%
839	16370	90.01%	0.15%	0.03%	9.81%
840	9205	80.40%	2.59%	0.15%	16.86%
842	11890	87.53%	0.15%	0.51%	11.81%
Média:	-	81.80%	1.00%	3.82%	13.37%
Mediana:	-	82.12%	0.16%	1.31%	12.01%

Tabela 2 – Resultado agregado relativo nos cenários em que houve discordância.

Nº do problema	Qtde. de códigos em discordância	Nº de Testes da MT	Nº de Testes desse projeto	MT falhou e atual passou	MT passou e atual falhou
736	2	22	8	0.00%	100.00%
742	234	100	5	99.15%	0.85%
744	17	101	4	41.18%	58.82%
751	1969	5	8	0.00%	100.00%
798	378	5	5	34.66%	65.34%
800	718	10	7	44.29%	55.71%
804	137	12	9	75.91%	24.09%
806	628	5	9	73.25%	26.75%
807	636	24	9	39.78%	60.22%
809	12	10	5	0.00%	100.00%
810	390	10	3	0.00%	100.00%
811	837	10	9	8.72%	91.28%
812	618	10	3	0.00%	100.0%
815	11	10	7	0.00%	100.00%
816	1625	7	7	23.45%	76.55%
817	262	10	6	18.32%	81.68%
819	41	20	8	0.00%	100.00%
820	1408	9	9	0.21%	99.79%
821	350	9	4	0.00%	100.00%
822	497	16	7	6.04%	93.96%
823	355	11	6	19.15%	80.85%
824	1259	10	4	95.47%	4.53%
827	196	10	6	99.49%	0.51%
828	1511	20	5	1.79%	98.21%
829	240	10	5	10.42%	89.58%
831	1136	10	6	1.58%	98.42%
832	79	21	4	91.14%	8.86%
833	10	12	11	0.00%	100.00%
834	22	10	8	0.00%	100.00%
835	137	4	9	3.65%	96.35%
836	179	3	9	21.79%	78.21%
838	22	10	7	18.18%	81.82%
839	29	10	7	82.76%	17.24%
840	252	10	9	94.44%	5.56%
842	79	15	9	22.78%	77.22%
Média:	-	-	-	28.95%	71.05%
Mediana:	-	-	-	18.18%	81.82%

4.1 CONJUNTO DE TESTES DA MACHINE TEACHING COM DESEMPENHOS MELHORES

A análise de cada exercício individualmente foi feita olhando as porcentagens dos grupos de testes originais e desse projeto, e averiguando quais possíveis cenários não foram levados em consideração para cada um deles. Oito dos problemas tiveram uma porcentagem nos cenários "FP", onde os códigos falharam em pelo menos um teste da Machine Teaching mas passaram em todos desse projeto, maior do que nos cenários "PF", onde os códigos passaram em todos os testes da Machine Teaching mas falharam em pelo menos um dos testes desse projeto. Pra tentar entender melhor porque os nossos testes não desempenharam tão bem em alguns casos, apresentamos uma análise a seguir.

Problema 742

Enunciado. *Escreva uma função definida por **substitui(s, x, i)** que receba uma string *s*, um caractere *x* e um número inteiro *i* entre 0 e o comprimento da string, e retorne uma string igual a *s*, exceto que o elemento da posição *i* deve ser substituído pelo caractere *x*.*

Esse exercício trata de manipulação de strings. Observamos que o conjunto de testes original da ferramenta continha 100 casos de teste, ao passo que o projeto por particionamento de entrada levou a apenas 5 casos de teste. Algumas características que não estavam sendo tratadas pelos nossos particionamentos foram: substituição no fim da string, substituição em letras repetidas, substituição em um índice específico. Analisando o particionamento proposto no atual projeto, podemos ver que efetivamente havia um bloco que levava a substituições na primeira posição da string, mas não havia um bloco que levasse a uma substituição na última posição da string. Isso pode ser visto como um defeito no particionamento proposto, que poderia ser melhorado. Outro tipo de defeito percebido foi em substituições em strings com letras repetidas. Esse tipo de defeito dificilmente seria antecipado pelo projetista dada a especificidade do problema; porém, como a plataforma Machine Teaching conseguiu captar esse defeito no seu conjunto de 100 casos de testes. Seria necessário que o caso de teste usasse uma string com letra repetida e a substituição não ser na primeira ocorrência dessa letra. O código 1 é um exemplo de código que só falharia nesse tipo de caso de teste.

```
def substitui(s,x,i):
    return str.replace(s,s[i],x,1)

substitui('dragagem', 'o', 5)
```

Código 1 – Código defeituoso no problema 742 e o caso de teste.

Problema 804

Enunciado. *Faça uma função chamada **filtra_pares** que receba uma tupla com quatro elementos inteiros como parâmetro, e retorne uma nova tupla contendo apenas os elementos pares da tupla original, na mesma ordem em que se encontravam. Esse tipo de operação onde se selecionam elementos de um conjunto inicial que satisfazem uma determinada propriedade é bastante comum em computação, e se chama **filtragem**.*

Esse problema trata de manipulação de tuplas. Aqui um fenômeno interessante ocorre: os testes cadastrados na plataforma fornecem como entrada uma lista ao invés de tupla, e muitos códigos falharam pois usaram fatiamento para obter os valores. Um exemplo de código funcionalmente correto, mas reprovado pelos testes da Machine Teaching é mostrado em 2. Esse código é reprovado pois tenta concatenar uma lista obtida no fatiamento com a tupla `filtragem`.

```
def filtra_pares(tuplas4):
    filtragem = ()
    for i in range(len(tuplas4)):
        if tuplas4[i]%2==0:
            filtragem += tuplas4[i:i+1]
    return filtragem
```

Código 2 – Código correto no problema 804 que está sendo reprovado.

Como esse problema não foi de particionamento, uma segunda execução foi feita após os testes obtidos da Machine Teaching serem modificados para darem como entrada tuplas ao invés de listas. Com isso, o número de códigos em discordância caiu de 137 para 70, com uma porcentagem relativa de 61.43% para códigos que reprovaram nos testes da Machine Teaching e foram aprovados nos desse projeto contra 38.57% para códigos que foram aprovados nos testes da Machine Teaching e reprovados nos dos desse projeto. Dessa vez, o principal defeito observado foram em códigos que enumeravam todas as tuplas possíveis a serem retornadas e acabavam falhando, esquecendo alguma combinação possível. O código 3 é um exemplo que representa praticamente todos os códigos que falharam. Esse tipo de defeito poderia ter sido captado pelos testes desse projeto caso tivesse sido levado em consideração a quantidade de valores pares e a posição desses valores na tupla durante o projeto de teste, apesar de aumentar a quantidade de casos de testes.

```
def filtra_pares(tupla):
    num1, num2, num3, num4 = tupla
    n1 = num1%2
    n2 = num2%2
    n3 = num3%2
```

```
n4 = num4%2
if (n1 == 0 and n2 == 0 and n3 == 0 and n4 == 0):
    return num1, num2, num3, num4
elif (n1 != 0 and n2 ==0 and n3 == 0 and n4 == 0):
    return (num2, num3, num4)
elif(n1 == 0 and n2 !=0 and n3 != 0 and n4 != 0):
    return (num1,)
elif(n1 == 0 and n2 ==0 and n3 != 0 and n4 != 0):
    return (num1, num2)
elif(n1 != 0 and n2 !=0 and n3 == 0 and n4 != 0):
    return (num3,)
elif(n1 == 0 and n2 !=0 and n3 == 0 and n4 != 0):
    return (num1, num3)
elif(n1 == 0 and n2 !=0 and n3 == 0 and n4 == 0):
    return (num1, num3, num4)
elif(n1 == 0 and n2 !=0 and n3 == 0 and n4 == 0):
    return (num1, num4)
elif(n1 == 0 and n2 ==0 and n3 == 0 and n4 != 0):
    return (num1 , num2, num3)
elif(n1 != 0 and n2 == 0 and n3 != 0 and n4 != 0):
    return (num2,)
elif(n1 != 0 and n2 ==0 and n3 == 0 and n4 != 0):
    return (num2, num3)
elif(n1 != 0 and n2 ==0 and n3 != 0 and n4 == 0):
    return (num2, num4)
elif(n1 != 0 and n2 !=0 and n3 == 0 and n4 == 0):
    return (num3, num4)
elif(n1 != 0 and n2 !=0 and n3 != 0 and n4 == 0):
    return (num4,)
else:
    return ()
```

Código 3 – Exemplo de código defeituoso no problema 804.

Problema 806

Enunciado. *(...) Escreva uma função chamada **colisao** que, dados dois retângulos, determine se eles se interceptam ou não. Cada retângulo é determinado pelas coordenadas x e y de dois de seus vértices diametralmente opostos, representando a diagonal que vai*

da esquerda para a direita e de baixo para cima. Os lados de cada retângulo são sempre paralelos aos eixos x e y. (...)

Esse problema também envolve manipulação de tuplas. O que ocasionou uma taxa mais alta foram os testes cujos valores das coordenadas "x" e "y" eram diferentes, um fator que não foi levado em consideração durante a criação dos particionamentos.

Problema 824

Enunciado. *Faça uma função chamada **uppCons** que receba como entrada uma frase e retorne a frase com todas as suas consoantes em maiúsculas (e os demais caracteres exatamente como estavam na frase original).*

Esse problema envolve manipulação de strings. Nesse caso, não foi levado em consideração situações em que a entrada "frase" contém vogais acentuadas, algo que foi captado nos casos de testes da Machine Teaching. Alguns códigos acabavam ao invés de checar se uma determinada letra era consoante, checava se ela não era vogal esquecendo das possibilidades de vogais acentuadas, o que acabava tratando vogais acentuadas como consoante e as tornando maiúsculas caso fossem minúsculas.

Problema 827

Enunciado. *Faça uma função chamada **qtd_divisores** que conte quantos divisores um número tem. Este número será passado como entrada.*

Exemplo: Se o número for 10, os divisores são: 1, 2, 5 e 10; total de 4 divisores.

Nesse exercício, é entregue um número inteiro e se espera como retorno a quantidade de divisores desse número. Aqui, houve mais falhas nos testes da Machine Teaching devido a um teste correspondente ao número zero, algo que não foi implementado nos testes novos. O particionamento desse projeto falhou ao não considerar o número zero separadamente.

Problema 832

Enunciado. *Faça uma função booleana chamada **eh_quadrada** para identificar se uma matriz é quadrada. Observação: uma matriz vazia (sem nenhuma linha nem coluna) é considerada quadrada.*

Esse problema envolve manipulação de listas. Nesse caso, matrizes com uma linha e várias colunas ou com múltiplas colunas e linhas tiveram problemas em códigos que passaram nos testes desse projeto. Os códigos dos alunos nesse problema não tiveram um defeito comum em específico, assim o projetista dificilmente teria conseguido criar particionamentos que captassem esses defeitos. É mostrado nos códigos 4, 5, 6, exemplos de submissões defeituosas feitas pelos alunos.

```
def eh_quadrada(mat):
    if any(len(linha) != len(mat) for linha in mat):
        return False

    for i in range(len(mat)):
        for j in range(i):
            if mat[i][j] != mat[j][i]:
                return False
    return True
```

Código 4 – Exemplo 1 de código defeituoso para o problema 832.

```
import math
def eh_quadrada(x):
    return math.sqrt(len(x)) == int(math.sqrt(len(x)))
```

Código 5 – Exemplo 2 de código defeituoso para o problema 832.

```
def eh_quadrada(matriz):
    contador = 0
    if len(matriz) == 0:
        return True
    for item in matriz:
        if len(item) == len(matriz):
            contador += 1
        if contador == len(matriz):
            return True
        else:
            return False
```

Código 6 – Exemplo 3 de código defeituoso para o problema 832.

Problema 839

Enunciado. *Um grupo de amigos deseja fazer uma viagem e decidiram ir de carro. Pelas regras rodoviárias um veículo convencional tem a capacidade de transportar até 5 passageiros, porém há veículos com outras capacidades. Construa uma função em Python chamada **carros** para calcular e retornar o número exato de carros necessários para esta viagem, considerando que seja dado como entrada o número de pessoas. Caso os veículos considerados sejam de capacidades não convencionais, será dado também como entrada a capacidade dos veículos, considerando que todos os veículos são iguais.*

Esse problema tem como objetivo treinar parâmetro padrão e uso de arredondamento. Olhando alguns exemplos, um particionamento que poderia ter sido feito seria um relativo a uma quantidade exata de pessoas para a capacidade do carro, ou seja, número de pessoas é múltiplo da capacidade ou não é múltiplo da capacidade.

Problema 840

Enunciado. *João deseja fazer bolos para seus amigos, usando uma receita que indica que devem ser usadas 2 xícaras de farinha de trigo, 3 ovos e 5 colheres de sopa de leite. Em casa ele tem **A** xícaras de farinha de trigo, **B** ovos e **C** colheres de sopa de leite. João não tem muita prática com a cozinha, e portanto ele só se arriscará a fazer medidas exatas da receita de bolo (por exemplo, se ele tiver material suficiente para fazer mais do que 2 e menos do que 3 bolos, ele fará somente 2 bolos). Sabendo disto, ajude João escrevendo uma função chamada **bolos** que determine qual a quantidade máxima de bolos que ele consegue fazer. (...)*

Aqui, as maiores falhas ocorreram quando um ou mais valores não são múltiplos e/ou quando um dos ingredientes tem uma quantidade bem superior aos outros. Inicialmente, os testes da Machine Teaching eram feitos de forma aleatória para esse problema, porém esses defeitos foram observados e foi feito um projeto de teste parar criar casos que verificam esse defeito onde um ingrediente está compensando a quantidade de outro. Nesse caso, os testes da Machine Teaching já foram corrigidos para pegar alguns desses defeitos.

Discussão

Apesar de não usarmos o desempenho dos testes do grupo anterior para comparação, também foram identificadas taxas mais altas nos cenários em que os códigos falharam nos testes deles e passaram nos da Machine Teaching do que no cenário contrário em 21 dos 34 problemas, mesmos os testes sendo mais atuais. Já em relação aos testes desse projeto, houve uma diferença, com os testes do grupo anterior tendo desempenho melhor em 8 dos 34 problemas, e desempenho igual em 2. Ainda assim, houve alguns casos em que os testes foram melhor instanciados como por exemplo o problema 816, onde os códigos que falharam nos testes do grupo anterior e passaram nos desse projeto ficaram em 3.01% e no cenário contrário ficaram em 0.29%. Apesar dos particionamentos serem parecidos e a quantidade de testes desse projeto ser maior, a instanciação dos valores nos casos de testes do projeto atual não foi boa nesse problema em específico.

Com relação a quantidade de testes, esse projeto optou por uma quantidade razoavelmente menor de casos, assumindo uma postura de exercer o mínimo de esforço durante a criação de teste, como forma de simular uma postura de menor "dispersão de energia" ao demonstrar a eficácia do método de particionamento de entrada. Essa postura no entanto

não foi uma boa escolha pois, em algumas instanciações, a escolha dos valores deixava certos defeitos passarem. Uma escolha mais aleatória desses valores pode trazer benefícios de "cobrir" possíveis erros quando os particionamentos não foram suficientes (NTAFOS, 2001)

Apesar de em alguns casos os testes da Machine Teaching terem um desempenho melhor, é importante notar que muitos dos problemas continham quantidades superiores de testes, como nos problemas 742 e 744, que continham 100 e 101 casos de testes respectivamente. Essas grandes quantidades trazem o benefício de acabarem, por acaso, cobrindo algum tipo de defeito comum feito pelos alunos, mesmo que o projetista não tenha antecipado tal comportamento. Outro fator é fato dos casos de testes cadastrados na Machine Teaching evoluíram ao longo do tempo. O trabalho progressivo de descoberta de defeitos e melhoria dos casos de testes ocorre desde o início da plataforma. Se as execuções fossem feitas com os testes originais, é provável que o desempenho dos testes atuais fosse ainda melhor.

Mesmo assim, uma alternativa ao método Pairwise, o All Combinations, que cria um caso de teste para cada combinação possível dos particionamentos, poderia ser utilizado para ter uma cobertura ainda melhor dos defeitos possíveis, ao custo de ter mais casos de teste. Apesar desse custo, ainda sim seria um experimento válido, dado que é de extrema importância não aprovar códigos defeituosos pois esses deixariam o aluno com a impressão de um código correto quando ele não está.

Outra coisa a levar em consideração é o fato de ser possível que alguns dos conjuntos de testes elaborados acharam mais defeitos por serem mais criteriosos dado que não existe especificação formal. Como dito anteriormente, a determinação do que deve ser avaliado é baseada nas características que o projetista de teste considerou relevante, ou seja, sua interpretação do enunciado. Isso pode afetar a replicabilidade do experimento devido a seleção de características diferentes (BALDWIN, 2018, cap. 7).

Ainda assim, mesmo com todas essas modificações, muitos problemas encontrados durante o projeto ocorreram devido a falta de cobertura de alguma característica das entradas durante a criação dos particionamentos ou a má instanciação dos valores nos casos de testes. Ao mesmo tempo, alguns dos defeitos detectados podem ser atribuídos a uma má interpretação dos enunciados ou a enunciados mal elaborados, já que os enunciados muitas vezes são escritos com um contexto de mundo real o que cria certa ambiguidade. Um exemplo é o problema 811 que trata sobre entradas relativas às dimensões de um colchão. Não se sabe se o projetista original realmente quis dizer um colchão real ou um paralelepípedo qualquer.

5 RECOMENDAÇÕES PARA CRIAÇÃO DE ENUNCIADOS E PROJETO DE TESTE

Nesta seção é divulgado um pequeno compilado de informações que podem ajudar o professor na transmissão da tarefa a ser feita, e ao projetista durante o projeto de teste, ambas conseguidas por meio da análise de falhas e defeitos das submissões dos alunos.

5.1 RECOMENDAÇÕES PARA CRIAÇÃO DE ENUNCIADOS

A criação do enunciado é importante na hora da verificação automática através de testes. Assim, é importante ressaltar que caso o objetivo do professor seja mensurar ou incentivar o processo de pensamento crítico do aluno sobre o problema é recomendado que algumas ou todas as informações aqui dadas sejam deixadas de lado, podendo tais dicas serem reduzidas de acordo com o progresso do aluno no curso. De todas as formas, o enunciado pode ser mais genérico, ou seja, com uma terminologia mais informal; o que o torna mais ambíguo, exigindo do aluno um mapeamento do mundo real ao mundo da programação. Ou pode ser mais preciso, com uma terminologia mais formal, poupando o aluno de fazer tal mapeamento o que pode não ser pedagogicamente interessante.

1º: Exemplo base

Um pequeno exemplo do tipo de entrada que a solução receberá e que deverá emitir diminui bastante a incerteza com relação ao exercício. Em geral, para enunciados mais complexos, a simples ilustração de um exemplo de entrada e saída pode clarear alguns resquícios de dúvidas.

> Enunciado original: Considere que *a* e *b* são duas strings à escolha do usuário. Faça uma função, chamada **concatenacao**, que retorne a concatenação delas no formato *abba*.
>
> Enunciado modificado: Considere que *a* e *b* são duas strings à escolha do usuário. Faça uma função, chamada **concatenacao**, que retorne a concatenação delas no formato *abba*. Exemplo:
> `> > > concatenacao(‘‘ga’’,’’to’’)`
> `gatotoga`

2º: Exemplo extra

Um exemplo de um caso não trivial pode incentivar o aluno a pensar criticamente. Novamente, é importante o professor saber o quanto ele quer que o aluno pense sobre o

problema. Se o professor quer incentivar o aluno a raciocinar sobre o exercício e os casos menos triviais, esse exemplo deve ser omitido.

> Enunciado original: Considere que *a* e *b* são duas strings à escolha do usuário. Faça uma função, chamada **concatenacao**, que retorne a concatenação delas no formato *abba*.
>
> Enunciado modificado: Considere que *a* e *b* são duas strings à escolha do usuário. Faça uma função, chamada **concatenacao**, que retorne a concatenação delas no formato *abba*. Exemplo:
>
> ```
> > > > concatenacao(‘‘ga’’,’’to’’)
> gatotoga
> > > > concatenacao(‘‘mas tem’’,’’ 10’’)
> mas tem 10 10mas tem
> ```

3º: Restrição de tipo

Especificar os tipos concretos dos dados ao invés de um genérico facilita ao aluno saber quais operações ele pode aplicar. Caso o professor queira utilizar tipos que não necessariamente tem representação direta na linguagem Python de forma que o aluno pense na melhor forma de tratar a solução, essa restrição pode ser omitida.

> Enunciado original: Faça uma função **retira_pontuacao** que, dada uma frase, retorne a frase onde todos os caracteres de pontuação (incluindo travessão, vírgula, dois pontos, ponto e vírgula, além da pontuação de encerramento de frase) tenham sido substituídos por espaço.
>
> Enunciado modificado: Faça uma função **retira_pontuacao** que, dada uma string, retorne uma string onde todos os caracteres de pontuação (incluindo travessão, vírgula, dois pontos, ponto e vírgula, além da pontuação de encerramento de frase) tenham sido substituídos por espaço.

Nesse caso em específico a troca de "frase" por "string" foi feita como forma de não limitar o aluno a somente frases gramaticalmente corretas. Por exemplo, a entrada pode ter pontuação em posições gramaticalmente incorretas? Ela pode ter símbolos ou dígitos fora de contexto? Palavras sem nexo? Múltiplos espaços consecutivos? Essas são perguntas que os alunos poderiam assumir uma resposta quando a entrada é dita como "frase" e outra se a entrada fosse dita como "string". Mesmo assim, não resolveu todos os problemas pois ainda não se sabe, por exemplo, se uma entrada pode ser vazia ou não.

4º Restrições adicionais

Especificar tamanho, intervalo (se for aberto ou fechado), composição dos dados, etc. Essas restrições são úteis caso queira especificar tamanhos máximos e mínimos para dados atômicos ou estruturados que podem não ser triviais. Também pode-se especificar tempo máximo de execução, caso seja relevante.

> Enunciado original: Escreva uma função definida por **substitui(s, x, i)** que receba uma string *s*, um caractere *x* e um número inteiro *i* entre 0 e o comprimento da string, e retorne uma string igual a *s*, exceto que o elemento da posição *i* deve ser substituído pelo caractere *x*.
>
> Enunciado modificado: Escreva uma função definida por **substitui(s, x, i)** que receba uma string *s*, um caractere *x* e um número inteiro *i* entre 0 e o comprimento da string, e retorne uma string igual a *s*, exceto que o elemento da posição *i* deve ser substituído pelo caractere *x*.
> Entradas:
> `s: string e 1 <= len(s)`
> `x: string e len(x) == 1`
> `i: int e 0 <= i <= len(s) - 1`
>
> Enunciado modificado 2: Escreva uma função definida por **substitui(s, x, i)** que receba uma string não vazia *s*, um caractere *x* e um número inteiro *i* entre 0 (inclusive) e o comprimento da string (exclusive), e retorne uma string igual a *s*, exceto que o elemento da posição *i* deve ser substituído pelo caractere *x*.

5º Dica

Uma dica seria qualquer informação que seja útil porém não possa ser deduzida diretamente do enunciado como uma função que deva ser usada ou um caso não trivial que seja preferido não fornecer exemplos. Se tal dica é relacionada a uma função, é ideal que seja explicitada no exemplo e/ou indicada uma fonte ou breve síntese de como ela funciona.

> Enunciado original: Faça uma função chamada **media_matriz** que dada uma matriz de inteiros não vazia, retorna a média de todos os números da matriz (com exatamente duas casas decimais de precisão).
>
> Enunciado modificado: Faça uma função chamada **media_matriz** que dada uma matriz de inteiros não vazia, retorna a média de todos os números da matriz (com exatamente duas casas decimais de precisão).

Dica: Utilize a função round().
Leia sobre em: https://docs.python.org/3/library/functions.html#round

Enunciado modificado 2: Faça uma função chamada **media_matriz** que dada uma matriz de inteiros não vazia, retorna a média de todos os números da matriz (com exatamente duas casas decimais de precisão).
Dica: Utilize a função round(numero, casas). Execute as funções round(2.351,2), round(2.355, 2) e round(2.359, 2) e depois tente implementá-la na sua função.

Agora, uma demonstração de um enunciado implementando todas as dicas acima:

Enunciado original: Faça uma função definida por **quant_palavras(frase)** que dada uma frase, retorne o número de palavras da frase. Considere que a frase pode ter espaços no início e no final.
Dica: veja a função split()

Enunciado modificado: Faça uma função definida por **quant_palavras(frase)** que dada uma frase, retorne o número de palavras da frase. Considere que a frase pode ter espaços no início e no final.
Entrada:
`frase: string e len(frase) >= 0`
As palavras podem ser compostas por letras, dígitos, e/ou símbolos (exceto espaços) porém isso não importa, trate todas como se fossem palavras.
Exemplo:

```
> > > quant_palavras(‘‘O guarda sol’’)
3
> > > quant_palavras(‘‘O Cais-115!  é o      primeiro!  ‘‘)
5
```

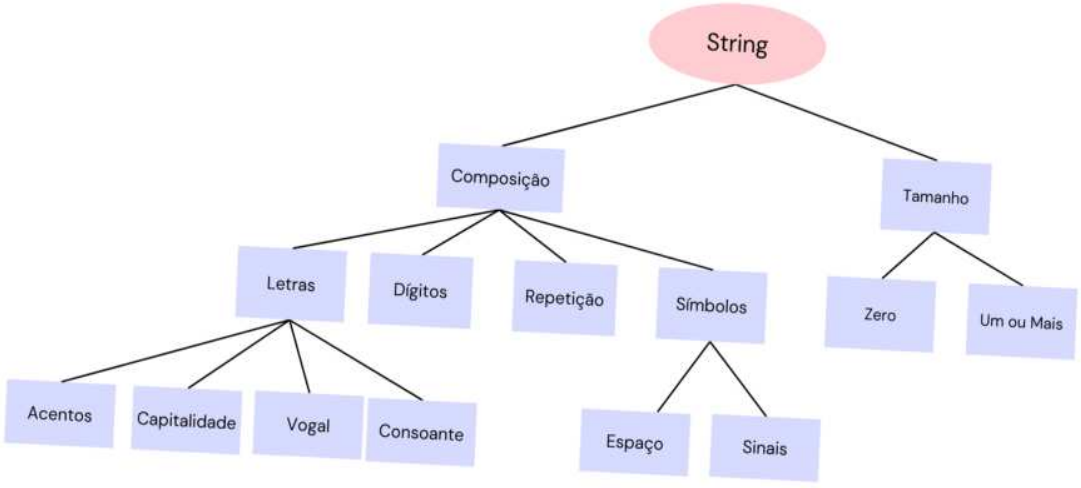
Dica: Utilize a função split().
Leia sobre ela em: https://docs.python.org/3.3/library/stdtypes.html#str.split
Dica 2: A string pode ser composta só por espaços.

5.2 RECOMENDAÇÕES PARA O PROJETO DE TESTES

Nessa seção, será relatado sobre dicas de como realizar o projeto de testes em conjunto com algumas observações feitas durante e após o mesmo, baseado nas experiências desse trabalho. Para o projeto de testes baseado em particionamento de entrada, observar o domínio das entradas é o passo principal. Com o objetivo de facilitar a visualização dos tipos de dados em Python, foram criadas as figuras 8, 9, e 10. Nelas, mostra-se alguns tipos de dados da linguagem Python em conjunto com algumas propriedades que eles podem ter. Elas servem como referência durante o projeto de testes ao projetista para

que ele possa observar o domínio da entrada dos testes e averiguar se um particionamento deve ser criado para alguma dessas propriedades, ou simplesmente para auxiliar durante a instanciação dos valores.

Figura 8 – Características típicas do tipo String nas legendas das imagens.



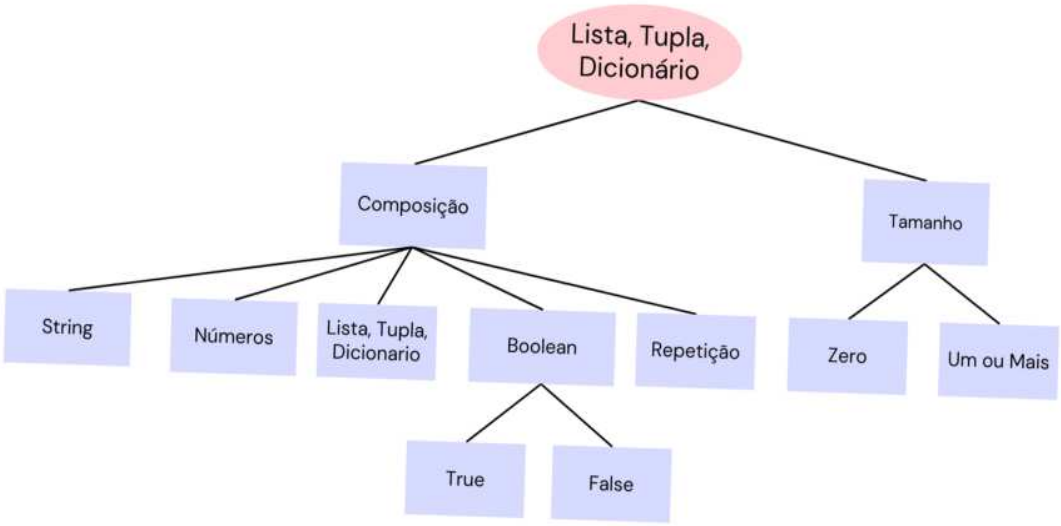
Elaboração: autor.

Além disso, também foram observados alguns defeitos em comum dentre os códigos dos alunos. Isso ressalta um detalhe importante: muitos desses erros cometidos pelos alunos ocorrem devido ao fato de estarem no início do processo de aprendizagem de programação. Assim, o projetista deve criar os testes com uma postura voltada a captar defeitos que um código criado por um programador experiente provavelmente não teria.

De todas as formas, abaixo está listada algumas áreas de pontos comuns de falhas que devem ser mantidas em mente durante o projeto de teste.

- Geral:
  - Testar valores nulos, ou de tamanho zero.
  - Testar estruturas com tamanhos pares e ímpares.
  - Testar valores de borda para o domínio de um valor qualquer, se possível.
  - Testar posição inicial, final, e na metade de dados estruturados, quando a posição fizer parte da estrutura.
- String:
  - Testar caracteres, símbolos e dígitos.

Figura 9 – Características típicas do tipo Lista, Tupla, e Dicionário, nas legendas das imagens.



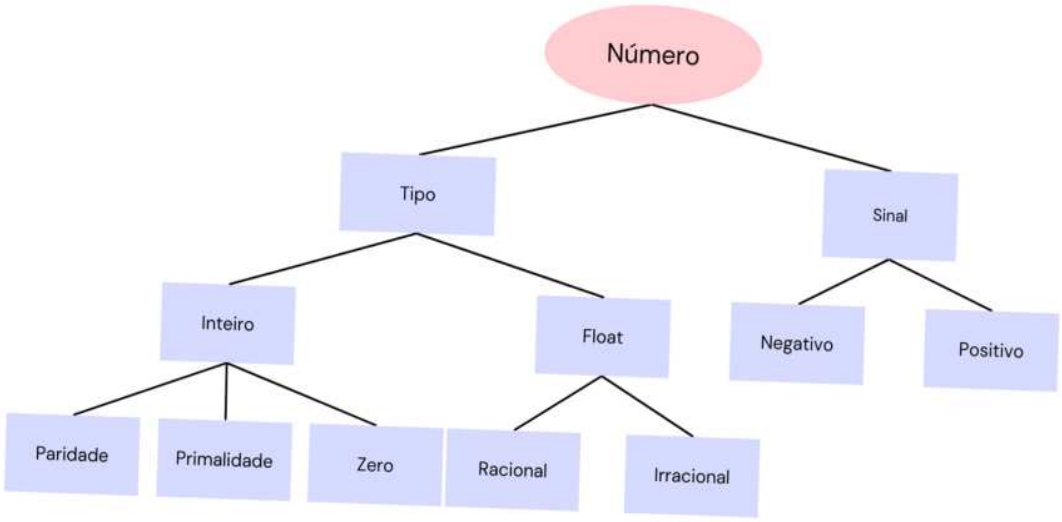
Elaboração: autor.

  - Em caracteres: testar pontuações, caracteres acentuados, maiúsculos, e minúsculos.
  - Testar string composta só por espaços e/ou múltiplos espaços consecutivos.
  - Testar string com múltiplos do mesmo caractere.

- Lista, tupla, e dicionário:
  - Se testando um elemento: testar lista/tupla/dicionário sem elementos, ou com múltiplas ocorrências do mesmo elemento;
  - Importante testar lista/tupla/dicionário que sejam compostas por outros elementos estruturados.
  - Usar variados valores de um mesmo tipo. Por exemplo: em uma lista de números, usar números de diferentes escalas, paridades, decimal e inteiro, etc.

- Números:
  - Testar inteiro e decimal, se permitido.
  - Testar o número zero.
  - Testar números menores que zero.
  - Testar números grandes (muitos dígitos).

Figura 10 – Características típicas do tipo Número nas legendas das imagens.



Elaboração: autor.

– Para problemas que envolvam arredondamento: testar números com mais casas decimais que as permitidas.

Outra dica para a elaboração dos testes, quando tratando de dados estruturados, é não limitar essas estruturas a tamanhos pequenos. No problema 832, por exemplo, que trata de manipulação de matrizes, foi observado que os testes da Machine Teaching e do grupo anterior continham um caso onde as matrizes eram 5x5 e 4x4 respectivamente enquanto a maior quadrada entre os casos de testes desse projeto era 1x1. Posteriormente, ao fazer um caso de teste com uma matriz 6x6, observou-se uma queda de 72 para 2 submissões, tornando o desempenho dos testes desse projeto superior. Outro defeito bem comum, quando lidando com coordenadas, são trocas de posições entre "x" e "y". Um caso de teste cujos valores das coordenadas forem iguais não vai conseguir detectar um defeito no código relacionado a esse caso. O exercício 834, a princípio, estava tendo um desempenho ruim nos testes desse projeto apesar do particionamento não mostrar problemas, após uma modificação dos valores foi observado que o erro estava na instanciação dos valores das coordenadas, que acabavam sendo iguais em vários testes. De maneira similar, é comum a troca entre as dimensões de matrizes, então é importante garantir que os testes explorem essa possibilidade.

Por último, na hora da elaboração dos testes, considerar o critério de combinação All Combinations, que utiliza todas as combinações possíveis durante o projeto de testes, para conseguir o máximo de cobertura possível dos defeitos.

6 CONCLUSÃO

A principal motivação para esse trabalho foi averiguar o quão efetiva é a aplicação do método de projeto de teste baseado em particionamento do espaço de entrada em problemas de programação voltada para alunos iniciantes. Uma segunda motivação foi a criação de um guia para ajudar os professores na elaboração dos enunciados e testes que eles queiram utilizar durante seus períodos de aula. Por último, como motivação final, foi demonstrar que o esforço utilizado para criar testes mais rigorosos é benéfico para fornecer um processo de aprendizagem mais sólido, dado que os alunos terão uma resposta mais realista sobre o quanto estão devidamente aprendendo durante o curso, já que códigos defeituosos não estão sendo aprovados, forçando-os a submeter códigos corretos.

Esse projeto utilizou dados fornecidos pela Machine Teaching para fazer uma nova análise do particionamento de entrada usando a amostra adquirida com foco em atestar se utilizar o método de particionamento de entrada demonstra um desempenho melhor na criação de teste mais completos. Esse desempenho foi aferido após a execução dos códigos dos alunos contra os testes criados nesse projeto e os cadastrados na Machine Teaching. A aferição consta a porcentagem de códigos que falharam nos cenários mencionados acima. Dessa forma, foi atestado que em 27 problemas dos 35 da amostra apontaram mais erros em códigos do que os encontrados pelos testes da Machine Teaching, com taxas de média e mediana mais altas.

Depois, foi averiguado quais os erros proeminentes dentro dos cenários, dessa vez incluindo os casos criados pelo grupo do projeto anterior (ALBUQUERQUE; COUTINHO; BOÉCHAT, 2023). Foi feita uma análise amostral dos casos e com isso foi obtida uma coletânea de erros que foram usadas para criar um guia de dicas voltadas a projeto de um enunciado que possivelmente propiciaria menos dúvidas sobre a solução do problema qual ele está descrevendo. Esse guia deve ser utilizado de acordo a discrição do lecionador, o quanto quer que o aluno interprete contra quanto o quer direcioná-lo. Além disso, foi criada uma lista de dicas sobre pontos prováveis de falha para auxiliar durante o projeto de testes. Contudo, existe a possibilidade do professor necessitar de apoio durante o projeto de teste para garantir uma boa qualidade dos mesmos. Um exemplo ocorreu durante o problema 834, onde inicialmente o desempenho dos testes criados por esse projeto estava muito inferior aos da plataforma Machine Teaching, porém, após uma revisão, notou-se que ocorreu problemas nas instanciações dos valores e, uma vez modificados e reexecutados, o desempenho dos testes melhorou e superou os da plataforma Machine Teaching.

6.1 TRABALHOS FUTUROS

Um possível trabalho a ser explorado no futuro é com relação à utilização de testes criados com particionamento de entrada no ambiente de aprendizagem e como isso afeta o processo de aprendizagem dos alunos. Como esse tipo de experimento iria requerer um processo mais extensivo como grupo de controle e experimental além de outras métricas de análise de aproveitamento dos alunos, toda inferência que pode ser feita com relação a isso se reduz à especulação. Contudo, é importante declarar que existem pesquisas na área de aprendizado que apontam um rendimento positivo no uso de um sistema de testes que quando apresentado com respostas errôneas fornece dicas específicas ao tipo de erro que foi recebido, dando dicas de como o erro pode ser resolvido (HALDEMAN et al., 2018). Tal resultado, somado ao fato de que para implementar a criação de testes usando particionamento de entrada é necessário a criação de requisitos que subdividem os domínios de entrada, uma possível melhoria do sistema no futuro seria um retorno significativo dado pelos requisitos os quais a solução do aluno não conseguiu cumprir.

Outro trabalho a ser tratado é com relação ao fato de alguns alunos, por terem acesso aos testes, se aproveitam dos mesmo para criar uma solução que vise somente os casos de testes específicos e não os requisitos do programa como um todo. Caso o professor não tenha tempo para acompanhar os códigos dos alunos individualmente, uma situação como essa propicia um ambiente de aprendizado falho, invalidando o propósito da ferramenta. Uma maneira de contornar esse problema seria utilizando o particionamento de entrada porém omitindo alguns testes de serem visualizados pelos alunos. Dessa forma, o aluno teria acesso aos requisitos necessários para alcançar o resultado de um teste, porém teria que criar a lógica mesmo assim pois haveria um teste oculto correspondente a combinação de requisitos que o teste que está sendo exibido supre.

E mais um trabalho possível, seria a verificação dos diferentes desempenhos quando usando critérios de combinação diferentes. Mais especificamente, quanto seria a diferença da cobertura entre casos de testes criados usando o All Combinations, ou seja, todas as combinações possíveis de particionamentos, em relação aos casos de testes projetados usando Pairwise.

REFERÊNCIAS

ALBUQUERQUE, A. de; COUTINHO, L.; BOÉCHAT, F. **MODELAGEM DE TESTES DE SOFTWARE. Uma Análise dos Resultados de Testes em Exercícios de Programação**. Dissertação (Trabalho de Conclusão de Curso) — Universidade Federal do Rio de Janeiro, 2023. Disponível em: http://hdl.handle.net/11422/21542.

AMMAN, P.; OFFUTT, J. **Introduction to Software Testing**. 2. ed. [S.l.]: Cambridge University Press, 2017.

BALDWIN, L. **Research Concepts for the Practitioner of Educational Leadership**. Leiden, The Netherlands: Brill, 2018. ISBN 978-90-04-36515-5. Disponível em: https://brill.com/view/title/38069.

COATES, H.; JAMES, R.; BALDWIN, G. A critical examination of the effects of learning management systems on university teaching and learning. **Tertiary Education and Management**, v. 11, n. 1, p. 19–36, 2005.

COX, N.; JONES, K. Exploratory data analysis. **Quantitative Geography, London: Routledge**, p. 135–143, 01 1981.

HALDEMAN, G. et al. Providing meaningful feedback for autograding of programming assignments. In: **Proceedings of the 49th ACM Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCSE '18), p. 278–283. ISBN 9781450351034. Disponível em: https://doi.org/10.1145/3159450.3159502.

KUHN, R. et al. Combinatorial testing: Theory and practice. **Advances in Computers, vol. 99**, 2015.

MORAES, L. O. et al. Machine teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação. **EduComp'22 (Online)**, 2022.

NTAFOS, S. On comparisons of random, partition, and proportional partition testing. **IEEE Transactions on Software Engineering**, v. 27, n. 10, p. 949–960, 2001.

OLIVEIRA, P. C. de; CUNHA, C. J. C. d. A.; NAKAYAMA, M. K. Learning management systems (lms) and e-learning management: An integrative review and research agenda. Universidade Federal de Santa Catarina, 2016.

APÊNDICE A – LINKS PARA O MATERIAL DO PROJETO

- A planilha com o agregado dos erros encontrados em conjunto com os seus respectivos problemas e informações está no link: https://docs.google.com/spreadsheets/d/1EjmRO7cK5ZQN4JMpMpfo17Wuo2F1oAGQCcn32m1ke6U/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 736 a 800: https://docs.google.com/spreadsheets/d/15m6PrCsUwXKAsoSFcl6XYRryOZ0AV_zsB-SUuaUs_IM/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 804 a 811: https://docs.google.com/spreadsheets/d/1eabTQY4eoATPfHF33bz0FKDtPc7CVMOAoR3L-exS0W8/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 812 a 820: https://docs.google.com/spreadsheets/d/1H5K16JDWbqZcFpoTYXQdMtkKG6DYPVfkPzn2p2fbpsg/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 821 a 828: https://docs.google.com/spreadsheets/d/1qVdOHf00RkXO5Kl-AzjogTeepp4SSWM_8Ds1csuP10/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 829 a 835: https://docs.google.com/spreadsheets/d/1g9cFaHmdWtjZpXaerTxAIuQZsOLNG6Jq7E6Z8VBev3M/edit?usp=sharing
- Planilha com a descrição, requisitos e casos de testes dos problemas de 836 a 842: https://docs.google.com/spreadsheets/d/1ATa9zuCX6GpnxUcFDNUdreE3RJwbdB0ab46k2dw5eak/edit?usp=sharing
- A pasta com as planilhas que contém os resultados das execuções estão no link: https://drive.google.com/drive/folders/1y0u1Z1PexsmLUTuLTYdCC1Bt5Q2DuQN6?usp=sharing
- O repositório que contém o código utilizado no projeto está em no link: https://github.com/Caesar66/tcc-input-partitioning