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

ARTHUR MENDONÇA SASSE

EXPLORANDO O AGRUPAMENTO DE CÓDIGOS PARA FEEDBACKS MAIS EFICIENTES NO ENSINO DE PROGRAMAÇÃO

RIO DE JANEIRO
2024

ARTHUR MENDONÇA SASSE

EXPLORANDO O AGRUPAMENTO DE CÓDIGOS PARA FEEDBACKS MAIS EFICIENTES NO 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: Profa. Carla Amor Divino Moreira Delgado
Co-orientador: Profa. Laura de Oliveira Fernandes Moraes
Co-orientador: Prof. Carlos Eduardo Pedreira

RIO DE JANEIRO
2024

Sasse, Arthur Mendonça
S252e Explorando o agrupamento de códigos para feedbacks mais eficientes no ensino de programação / Arthur Mendonça Sasse. – 2024.

54 f.

Orientadora: Carla Amor Divino Moreira Delgado.
Coorientadora: Laura de Oliveira Fernandes Moraes.
Coorientador: Carlos Eduardo Pedreira.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Ensino de programação. 2. Análise de dados educacionais. 3. Ambiente educacional. I. Delgado, Carla Amor Divino Moreira (Orient). II. Moraes, Laura de Oliveira Fernandes (Coorient). III. Pedreira, Carlos Eduardo (Coorient.). IV. Universidade Federal do Rio de Janeiro, Instituto de Computação. V. Título.

ARTHUR MENDONÇA SASSE

EXPLORANDO O AGRUPAMENTO DE CÓDIGOS PARA FEEDBACKS MAIS EFICIENTES NO 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 01__ de Fevereiro_____ de 2024_____

BANCA EXAMINADORA:

Profa. Carla Amor Divino Moreira Delgado
DSc (UFRJ)

Profa. Laura de Oliveira Fernandes Moraes
PhD (UFRJ)

Profa. Eldânae Nogueira Teixeira
DSc (UFRJ)

Profa. Carolina Gil Marcelino
PhD (CEFET-MG)

Dedico este trabalho ao meu avô Wilson.

AGRADECIMENTOS

Agradeço pelo apoio financeiro do CNPq, por meio da Bolsa de Iniciação Tecnológica em TICs (Processo 105600/2022-9), e do Instituto Reditus, por meio do Edital de Inovação 2022.

Agradeço à professora Laura Moraes, pelas oportunidades, pelas conversas e por todo o aprendizado. Agradeço à professora Carla Delgado, por todos os feedbacks e pela atenção, e ao professor Carlos Pedreira, por todo o trabalho, junto às professoras, para garantir o crescimento do projeto Machine Teaching. Agradeço aos colegas do projeto também, pela solidariedade e pela troca de experiências.

Agradeço à minha família e aos meus amigos de UFRJ, EJCM e CAInfo, por todo o suporte ao longo da graduação. Agradeço aos professores do IC pelo aprendizado e, em especial, aos professores João Paixão, Vinícius Gusmão, Maria Luiza Campos e Evandro Macedo, por despertarem em mim o fascínio por diferentes áreas da Computação. Por fim, agradeço a Larissa Galeno, minha companheira de vida pessoal e acadêmica, pelo apoio emocional, estratégico e técnico nessa jornada que compartilhamos.

"A Resposta à Grande Questão... Da Vida, o Universo e Tudo Mais... É... Quarenta e dois – disse Pensador Profundo, com uma majestade e uma tranquilidade infinitas."

O Guia do Mochileiro das Galáxias - Douglas Adams

RESUMO

Velocidade. Organização. Disponibilidade. Correções automáticas. São claros os benefícios das plataformas online de ensino de programação, que são válidos para turmas tradicionais de graduação até grandes cursos abertos, como aqueles presentes no *Coursera* ou *Codecademy*. Mas o feedback personalizado, além da aprovação ou reprovação nos casos de testes, é um elemento essencial do ensino que ainda depende da análise humana e não consegue ser reproduzido em larga escala nesses sistemas. Ao mesmo tempo, percebe-se, na prática, que muitos alunos, em suas respostas, incorrem nos mesmos erros ou padrões de resolução, que poderiam receber o mesmo tipo de feedback do professor. Este estudo é motivado pela perspectiva de usar as similaridades entre os programas submetidos pelos estudantes para reaproveitar as análises dos docentes e diminuir o esforço repetitivo. Dessa forma, este trabalho realiza uma pesquisa exploratória sobre a viabilidade do agrupamento automático de códigos como uma alternativa para reduzir o trabalho docente de produção de feedback para as atividades de prática em programação. Primeiro é apresentada uma análise das abordagens existentes para o agrupamento de códigos e reaproveitamento de feedbacks, além dos critérios para a escolha da ferramenta que passou por uma avaliação mais profunda. Uma vez escolhido o Overcode, o atualizamos para funcionar com o Python 3 e avaliamos a sua capacidade de clusterização em um conjunto de dados com mais de 100 questões e dezenas de milhares de soluções, extraídos do Machine Teaching, plataforma utilizada para o ensino de programação em cursos de graduação da Universidade Federal do Rio de Janeiro. Os resultados indicam que, usando a clusterização do Overcode, é possível reduzir significativamente o número de soluções para problemas de programação que um docente precisa avaliar em comparação com a abordagem tradicional, de avaliar os códigos de estudantes individualmente. Essa redução se mostrou ainda mais significativa para questões com grandes quantidades (centenas) de respostas. Além disso, para quantidades pequenas de questões (até 50) o Overcode mostrou-se uma ferramenta que não exige grande poder computacional, sendo executado em poucos segundos em um hardware de computador pessoal.

Palavras-chave: ensino de programação; análise de dados educacionais; ambiente educacional.

ABSTRACT

Speed. Organization. Availability. Automatic corrections. The benefits of online programming education platforms are clear, applicable to traditional undergraduate classes as well as massive open online courses (MOOCs) like those found on *Coursera* or *Codecademy*. However, personalized feedback, beyond simple pass or fail outcomes in test cases, remains a crucial element of teaching that still relies on human analysis and cannot be easily scaled in these systems. Simultaneously, it is observed in practice that many students make similar errors or follow similar patterns in their responses, which could potentially receive the same type of feedback from the instructor. This study is motivated by the prospect of utilizing similarities between programs submitted by students to leverage teachers' analyses and reduce repetitive effort. Thus, this work conducts an exploratory research on the feasibility of automatic code clustering as an alternative to reduce the teaching workload in providing feedback for programming practice activities. First, an analysis of existing approaches to code clustering and feedback reuse is presented, along with criteria for choosing the tool that underwent a more in-depth evaluation. Once Overcode was selected, we updated it to work with Python 3 and evaluated its clustering capability on a dataset with over 100 questions and tens of thousands of solutions extracted from Machine Teaching, a platform used for programming education in undergraduate courses at the Federal University of Rio de Janeiro. The results indicate that, using Overcode's clustering, it is possible to significantly reduce the number of solutions to programming problems that a teacher needs to assess compared to the traditional approach of evaluating individual student codes. This reduction was even more significant for questions with large quantities (hundreds) of responses. Additionally, for small quantities of questions (up to 50), Overcode proved to be a computationally lightweight tool, running in a few seconds on personal computer hardware.

Keywords: programming education; analysis of educational data; educational environment.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

As plataformas online de ensino de programação têm o potencial de escalar o acesso de estudantes a uma educação de qualidade por meio da disponibilização de materiais de estudo, exercícios e até correções automatizadas. Esse tipo de suporte permite otimizar o tempo do professor e também o tempo de estudo do aluno (PRICE; PETRE, 1997).

No entanto, um passo importante do processo de aprendizagem continua sendo um gargalo nesses sistemas: o feedback. No contexto específico do ensino de programação, exercícios práticos, como criação de programas ou componentes de programas, são muito comuns. Além de indicar se as respostas estão corretas ou não, o feedback didático para este tipo de atividade deve envolver principalmente: a identificação, por meio da detecção de erros nas respostas, de conceitos que não foram bem assimilados e sugestões de melhorias diretas para o código escrito pelos alunos (AMBROSE et al., 2010).

Cada tópico de programação pode ter vários exercícios e cada exercício, uma resolução diferente por parte de cada estudante. A solução de um aluno carrega suas próprias dúvidas e erros. Infelizmente é impossível um docente ter tempo suficiente para essa análise individual de todas as respostas quando possui uma turma muito grande. Nesse sentido, torna-se necessária uma ferramenta que torne mais eficiente esse aconselhamento para cada aluno, sem comprometer a personalização necessária para que o feedback seja efetivo (BUTLER; MORGAN, 2007), mesmo diante de uma multitude de respostas a serem analisadas.

De maneira mais concreta, se um professor ensina programação para 30 alunos e a cada semana passa 5 exercícios, então, semanalmente, ele ou seus monitores terão que analisar e produzir até 30 feedbacks para cada exercício e até 150 no total. Esse trabalho, além de cansativo, é repetitivo, já que muitos alunos repetem padrões, corretos ou errados, ao escrever seus códigos, para os quais os instrutores escrevem feedbacks semelhantes.

1.1 MOTIVAÇÃO

A ideia que motiva este trabalho é a de que: partindo do princípio que respostas semelhantes recebem feedbacks semelhantes, se fosse possível agrupar códigos que compartilham um mesmo padrão, então bastaria escrever um feedback para cada padrão, que serviria para todas as respostas agrupadas que seguem esse padrão. Seguindo o exemplo anterior, se num conjunto de 30 respostas para um exercício, fosse possível separar os códigos em 10 padrões, o instrutor teria que produzir apenas 10 feedbacks em vez de 30.

A partir dessa ideia, buscamos explorar a viabilidade desse tipo de abordagem para diminuir o trabalho repetitivo e, consequentemente, o tempo gasto pelos professores e monitores com a produção de feedback para respostas aos exercícios de programação.

1.2 OBJETIVO

Este trabalho visa conduzir uma pesquisa exploratória sobre a viabilidade de agrupar códigos dos alunos por padrões identificados e o uso desse agrupamento para tornar mais eficiente o processo de geração de feedbacks.

Primeiro analisamos as ferramentas disponíveis para clusterização de códigos semelhantes e produção mais eficiente de feedback, a partir de alguns critérios pré-definidos que guiaram a pesquisa. A ferramenta escolhida, o Overcode (GLASSMAN et al., 2015), foi analisada mais a fundo e adaptada para funcionar com a base de dados do Machine Teaching (MORAES et al., 2022), uma plataforma online utilizada para o ensino de programação em disciplinas de graduação da Universidade Federal do Rio de Janeiro. A exploração desses dados utilizando o Overcode foi guiada pelas seguintes perguntas de pesquisa:

1. O quanto o Overcode reduz a quantidade de códigos a serem analisados pelos professores?

2. Qual é o tempo de execução do Overcode para as questões do Machine Teaching? É um tempo que viabiliza sua utilização por professores?

As questões do Machine Teaching foram divididas em 2 grupos: questões com poucas respostas e questões com muitas respostas. As perguntas de pesquisa foram avaliadas em relação a todas questões e também de maneira individual para cada grupo. Para a primeira pergunta, foi utilizada como métrica a razão entre o número de agrupamentos formados e o total de respostas para cada pergunta. Já para a segunda, foram medidos os tempos de processamento da ferramenta para cada questão.

1.3 ORGANIZAÇÃO DO DOCUMENTO

No capítulo 2 é apresentada a fundamentação teórica, descrevendo em mais detalhes a plataforma Machine Teaching e os trabalhos relacionados ao nosso objetivo, que serviram como base para a escolha da ferramenta utilizada na exploração dos dados. No capítulo 3 é descrita a metodologia e sua execução, passando pela análise prévia da base de dados do Machine Teaching, pela formulação e utilização dos critérios para escolha da ferramenta entre os trabalhos relacionados até a adaptação da ferramenta escolhida e o detalhamento das métricas utilizadas na pesquisa. No capítulo 4 são apresentados os resultados, com gráficos e tabelas, em conjunto com uma discussão sobre eles. No capítulo 5, abordamos as conclusões, limitações do estudo e direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CORRETORES AUTOMÁTICOS

Corretores automáticos são sistemas de avaliação automatizada utilizados para corrigir resoluções de exercícios submetidas por estudantes (PRATHER et al., 2018). A principal vantagem oferecida por eles aos professores é reduzir o tempo necessário para realizar as correções. Ao mesmo tempo, possuem a limitação de avaliarem apenas a saída dos programas e não como eles foram construídos e como os conceitos ensinados foram aplicados. Por exemplo, dois programas podem ter a mesma saída e estarem corretos segundo o corretor automático, mas um deles pode apresentar problemas de estilo ou de boas práticas, que só conseguiriam ser detectados pela análise do próprio docente. Além disso, para códigos introdutórios, que costumam abordar problemas muito simples, o estudante pode recorrer a recursos simplórios para elaborar uma solução e não utilizar os recursos que deveriam ser praticados na atividade. Ou seja, o aluno pode tentar enganar o corretor automático propositalmente ou apenas estar realmente mal informado sobre o procedimento adequado na resolução da atividade.

No geral, esses sistemas funcionam da seguinte maneira: o usuário visualiza o enunciado do problema, escreve o código do seu programa num espaço de entrada do sistema e submete sua solução, que vai passar por um ou mais testes, indicando se está correta ou não.

2.1.1 Machine Teaching

O Machine Teaching é um exemplo de uso de correção automática, tendo sido utilizado como recurso didático de apoio às aulas práticas dos cursos introdutórios de programação oferecidos pelo Instituto de Computação da Universidade Federal do Rio de Janeiro (UFRJ) nos últimos 5 anos (MORAES et al., 2022). Entre os principais objetivos do Machine Teaching estão a coleta de dados durante a interação com a plataforma para o apoio à tomada de decisões relativas ao curso, o fornecimento de feedback imediato durante as atividades e a automação parcial da análise dos códigos dos alunos, facilitando o trabalho de acompanhamento e geração de feedback dos professores. Ou seja, além de uma ferramenta de suporte para o aprendizado dos alunos, também serve para a coleta e análise de dados educacionais (MORAES et al., 2022). Até o momento, o Machine Teaching já foi utilizado por mais de 140 turmas, 45 professores e 3900 alunos, somando mais de 560 mil interações em sua base de dados nos últimos 5 anos.

Nessa plataforma, as soluções são escritas em Python 3 e na forma de função com um nome pré-definido no enunciado, que deve retornar o valor esperado por cada caso de teste. Os resultados são apresentados de maneira individual para cada teste, indicando

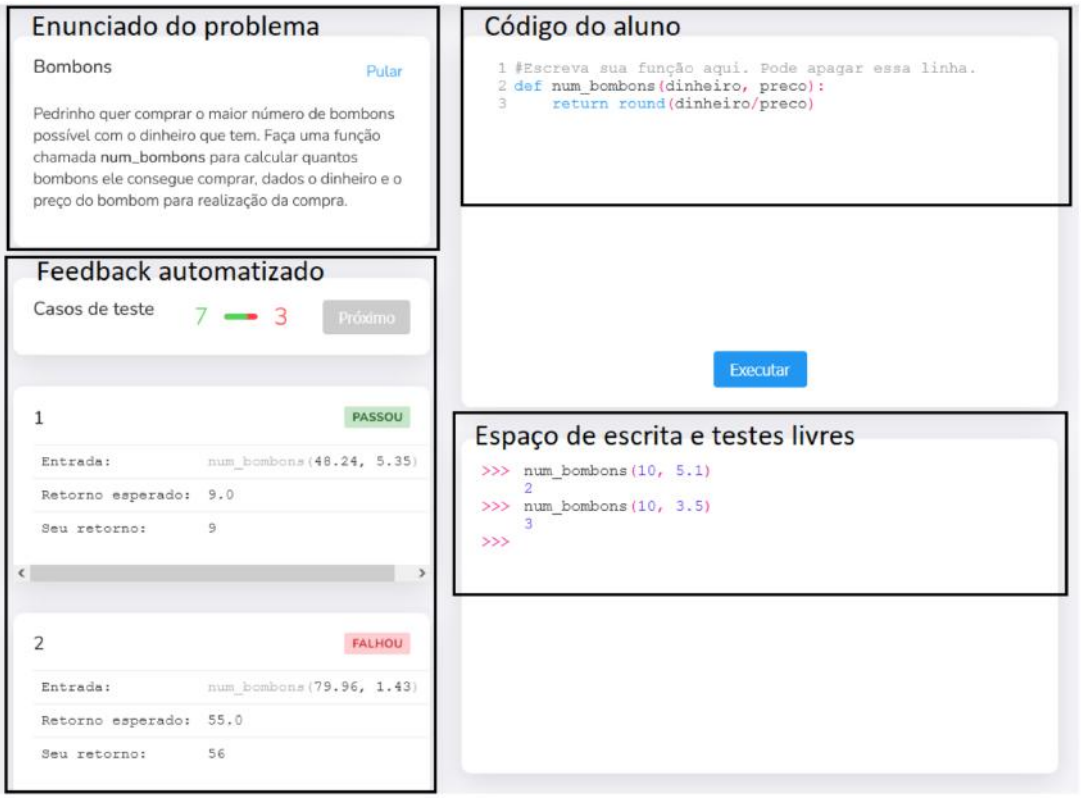

Figura 1 – Ambiente de desenvolvimento integrado do Machine Teaching

aprovação ou falha, junto com a entrada fornecida, o retorno esperado e o retorno do programa submetido. O aluno pode editar o código enviado e submetê-lo novamente, quantas vezes forem necessárias. A interface também dispõe de um espaço para escrever e executar os seus próprios casos de teste e pode ser vista em mais detalhes em Moraes et al. (2022).

2.2 TRABALHOS ANALISADOS

A seguir são apresentadas algumas ferramentas ou técnicas cujo objetivo está ligado à semelhança entre códigos, facilitação de feedback e/ou exploração de grandes quantidades de soluções de estudantes para problemas de programação. As ferramentas analisadas foram divididas em cinco métodos mais gerais:

- A **comparação de textos** foca no que diferencia um código de outro;
- Já a **detecção de plágio** busca similaridades entre programas que possam indicar plágio;
- O **code embedding** consiste na representação e manipulação de programas como vetores;
- Por sua vez a **síntese de programas** se baseia na geração automática de códigos a partir de especificações de alto nível;

- Por fim, as **árvores sintáticas** são formas de representar a estrutura de programas que podem ser utilizadas para uma comparação mais abstrata.

Um resumo desses trabalhos relacionados pode ser visto na Tabela 1.

Tabela 1 – Resumo dos trabalhos relacionados.

Ferramenta	Objetivo	Método
diff, meld, git diff	Destacar o menor número de mudanças para tornar 2 programas iguais.	Algoritmo para o problema da maior subsequência em comum entre 2 programas.
MOSS	Detectar plágio.	Comparação de fingerprints (hashes de partes dos programas).
STRANGE	Detectar plágio e explicar as semelhanças em linguagem natural.	Limpeza dos códigos e análise de trechos similares. As similaridades são explicadas por um modelo pré-definido de causas.
Code2Vec	Representar códigos como vetores que guardem suas propriedades semânticas.	Redes neurais que aprendem representações dos códigos a partir das árvores sintáticas abstratas.
ProtoTransformer	Automatizar feedback para problemas de programação, a partir de poucos exemplos.	Usar meta-aprendizagem e metadados das questões, para representar códigos de alunos e classificá-los de acordo com o feedback.
Singh, Gulwani e Solar-Lezama (2013)	Automatizar feedback para problemas de programação.	Algoritmo de síntese baseado em restrições e modelos de erros para cada problema.
Rivers e Koedinger (2013)	Automatizar feedback para problemas de programação.	Localizar cada solução em um espaço formado por todas as soluções possíveis para um problema.
FixPropagator e MistakeBrowser	Reutilizar feedbacks para códigos similares.	Agrupar códigos de estudantes pelas correções necessárias e propagar feedbacks para os códigos agrupados.
Codewebs	Buscar códigos similares a partir de uma árvore sintática fornecida.	Abordagem probabilística para verificar a equivalência entre subárvores sintáticas abstratas de cada código.
OverCode	Visualizar e explorar grandes quantidades de códigos.	Padronização dos códigos e clusterização baseada no conjunto de linhas de cada solução.
Huang et al. (2013)	Reutilizar feedbacks para códigos similares.	Agrupar soluções pela distância de edição em árvores entre programas.

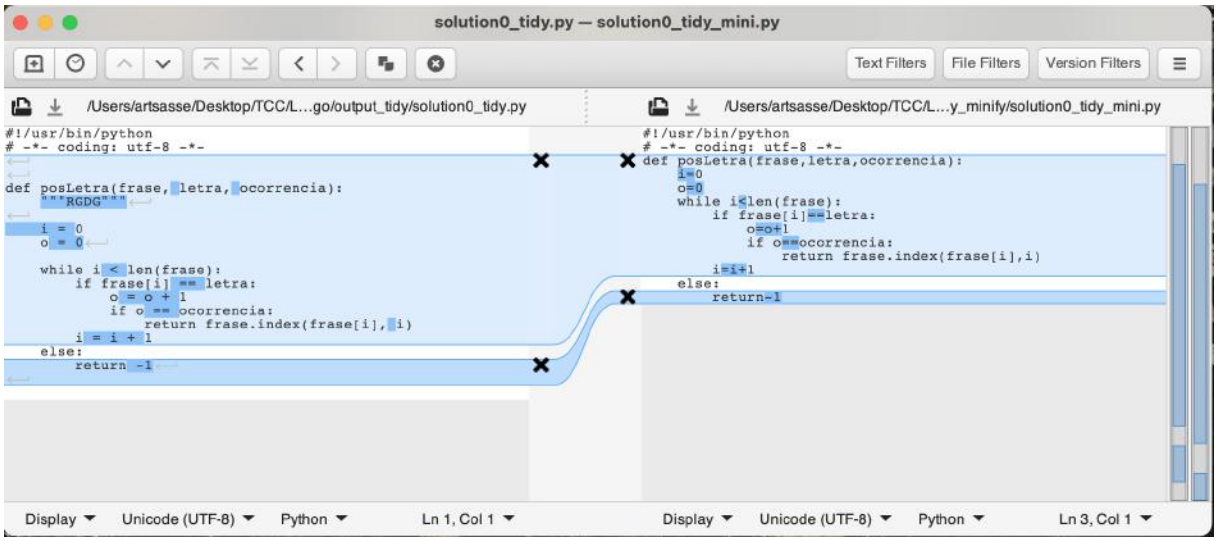

Figura 2 – Interface do *meld* (WILLADSEN; KENNEDY; LEGOLL, 2022).

2.2.1 Comparação de Texto

Ferramentas para a comparação de códigos já existem na computação há muito tempo, como o *diff*, um programa criado em 1975, como parte do sistema Unix, capaz de apontar o menor número de modificações necessárias nas linhas de um programa para torná-lo idêntico a outro (HUNT; MCILROY, 1976). Algumas ferramentas similares foram criadas posteriormente como o *meld* (WILLADSEN; KENNEDY; LEGOLL, 2022), representado na Figura 2, que traz uma interface visual para facilitar a análise e o *git-diff*, comando do programa *git* de versionamento de código, que mostra a diferença entre arquivos, commits ou ramificações de um repositório de código (GIT, 2023). No entanto, essas ferramentas não levam em conta similaridade sintática e semântica dos códigos, já que os programas são tratados como cadeias de caracteres das quais queremos extrair a maior subsequência em comum. Além disso, foram construídas com o objetivo de comparar dois elementos específicos e não buscar padrões em meio a um grande corpus de programas. Por isso, não se configuraram como ferramentas úteis para o interesse principal deste artigo.

2.2.2 Detecção de Plágio

A questão de buscar similaridade entre códigos também foi abordada por ferramentas como o MOSS (SCHLEIMER; WILKERSON; AIKEN, 2003), cujo objetivo é detectar plágio em soluções de tarefas de programação, como aquelas submetidas no Machine Teaching, mesmo quando há algum tipo de ofuscação como variáveis com nomes diferentes. Em complemento, Karnalim e Simon (2021) apresentam o sistema STRANGE, representado na Figura 3, que além de detectar plágios, também consegue explicar em linguagem natural, em inglês ou indonésio, quais são as similaridades entre os trechos selecionados.

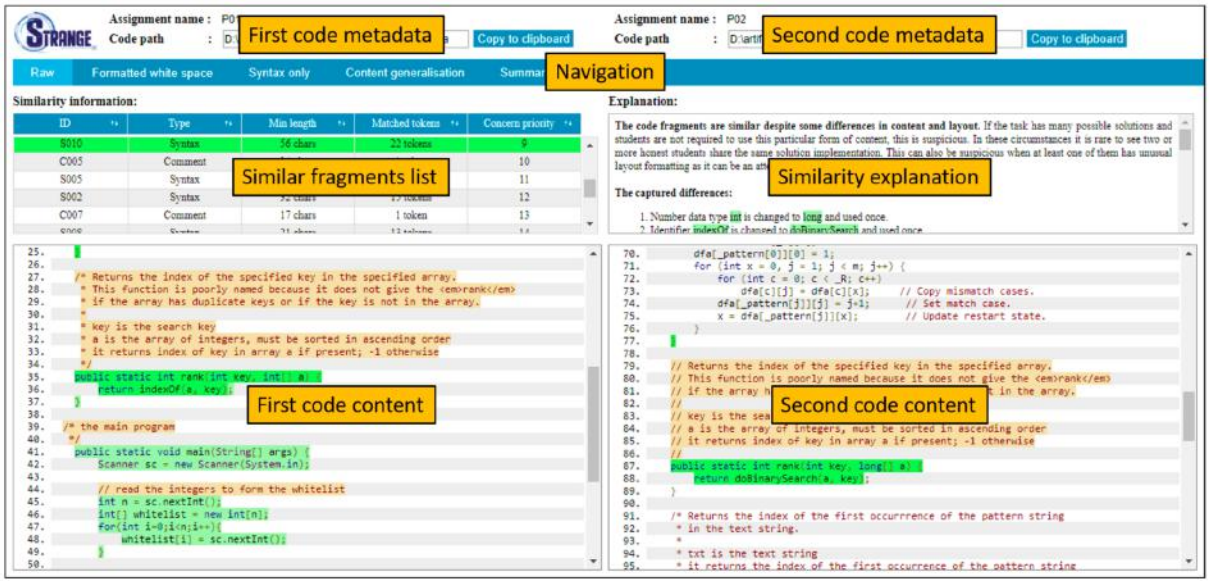

Figura 3 – Interface do STRANGE (KARNALIM; SIMON, 2021).

2.2.3 Code Embedding

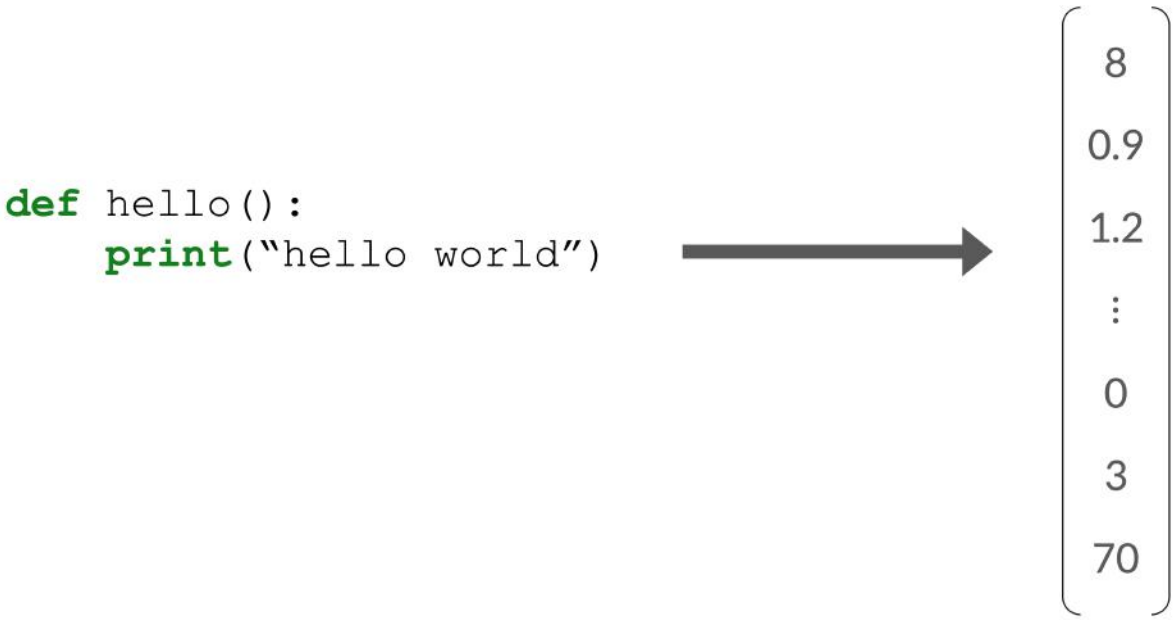

Figura 4 – Exemplo lúdico de transformação de um programa em vetor.

Alon et al. (2019) criaram um modelo neural, o "code2vec", que consegue representar trechos de código como vetores de distribuição contínua, como é exemplificado na Figura 4. Os vetores são construídos a partir das arvores sintáticas abstratas de cada trecho de código e podem ser utilizados para predizer características semânticas dos trechos, por exemplo, vetores próximos representam códigos com comportamento similar. Para demonstrar a eficácia dessa modelagem foi proposto o desafio de predizer o nome de um método a partir do vetor que representa seu código, tarefa na qual o code2vec obteve um desempenho 75% melhor do que outras alternativas testadas sobre a mesma base de dados.

Ao longo da tarefa foi percebido que essa representação em vetores conseguia capturar relações semânticas entre os códigos como similaridade, analogias e combinações.

Outra abordagem que se baseia no *code embedding*, é o ProtoTransformer (WU et al., 2021), que utiliza um processo de meta-aprendizagem. A meta-aprendizagem consiste em observar o desempenho de diferentes abordagens de aprendizado de máquina em uma gama de tarefas e, a partir dos metadados gerados, aprender novas tarefas de maneira mais rápida (VANSCHOREN, 2018). No caso do ProtoTransformer, o aprendizado é feito a partir de poucas amostras de feedback para cada problema de programação, além de informações complementares (enunciado e critérios de avaliação) sobre cada exercício, com o objetivo de gerar feedback automático para qualquer submissão futura. Esse método foi aplicado com sucesso na geração de feedback para mais de 16.000 submissões de estudantes de um curso de programação universitário.

2.2.4 Síntese de Programas

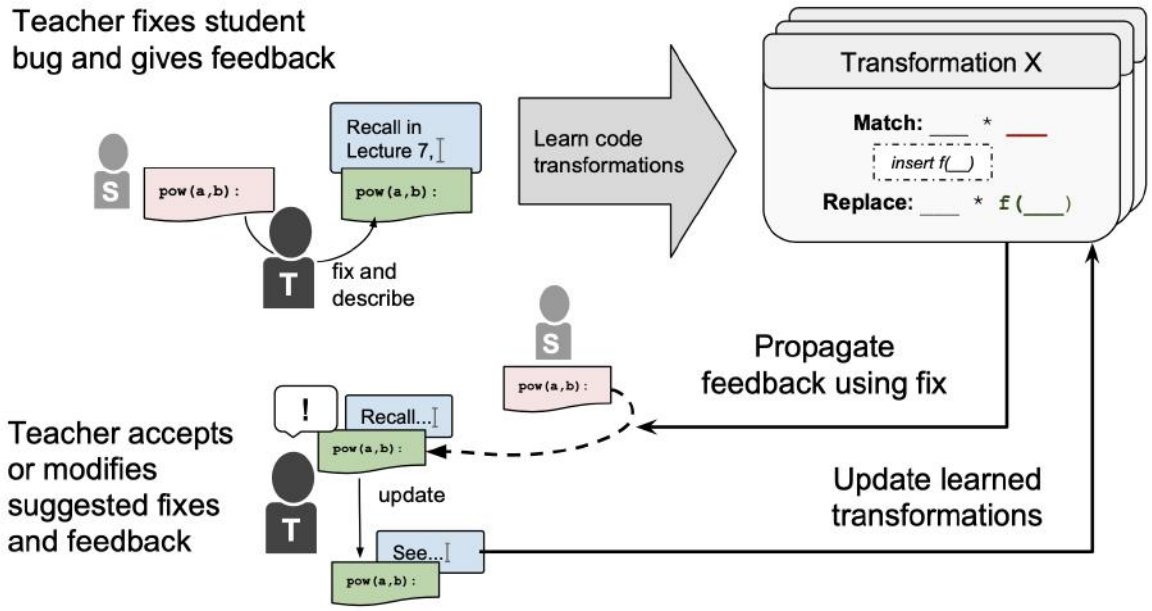

Figura 5 – Fluxograma do FixPropagator (HEAD et al., 2017).

A síntese de programas consiste na geração de programas a partir de especificações de alto nível (SOLAR-LEZAMA, 2008). No contexto da correção de exercícios de programação, algumas pesquisas utilizaram essa abordagem para gerar feedback individualizado e mais relevante para cada solução submetida. Entre elas destacam-se o trabalho de Singh, Gulwani e Solar-Lezama (2013), no qual o sistema, a partir de uma implementação de referência e um modelo de erros para cada questão, consegue derivar quais as correções mínimas para que o código de um aluno se torne correto. Isso serve de base para informar ao aluno o quão próximo ele está de acertar a questão e quais foram seus erros. Em

uma avaliação com milhares de alunos de cursos de programação do Massachusetts Institute of Technology (MIT), esse sistema, na média, conseguiu corrigir 64% das submissões incorretas.

Já o artigo de Rivers e Koedinger (2013) relata a experiência com uma abordagem orientada por dados, na qual a partir do enunciado de cada exercício, é definido um espaço de soluções com todos os caminhos possíveis até a resposta correta. Em seguida as submissões dos estudantes são projetadas nesse espaço, o que permite aferir o progresso de cada um e informar quais devem ser os próximos passos para alcançar a corretude.

Head et al. (2017) apresentam o *FixPropagator*, representado na Figura 5, e o *Mistake-Browser*, ferramentas que também utilizam a síntese de programas, baseadas nas correções feitas pelos professores e pelos próprios alunos, para agrupar soluções de estudantes pelas mudanças necessárias para torná-las corretas. Os feedbacks de um cluster são propagados para todas as submissões que fazem parte dele.

2.2.5 Árvores Sintáticas

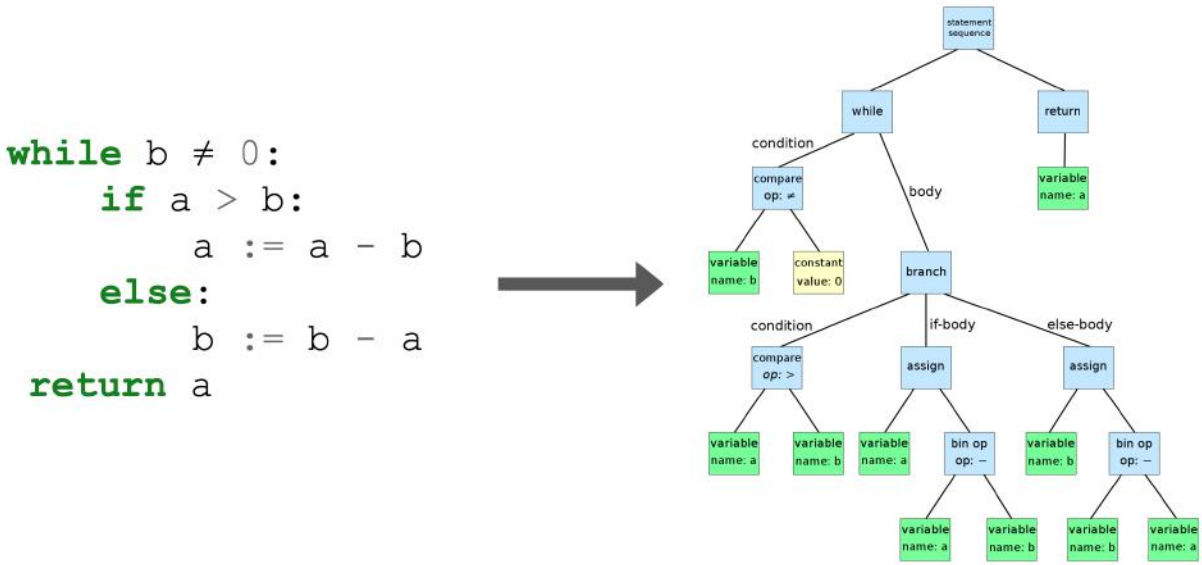

Figura 6 – Exemplo de Árvore Sintática Abstrata. Fonte: https://en.wikipedia.org/wiki/Abstract_syntax_tree (modificado).

O Codewebs (NGUYEN et al., 2014) parte da análise de árvores sintáticas abstratas (AST, do inglês *abstract syntax tree*), exemplificadas na Figura 6, para abordar o desafio de gerar feedbacks para milhares de submissões. O programa cria uma espécie de índice para os componentes de código que compõem as submissões. Os professores podem fazer pesquisas por códigos similares, mas para isso precisam montar manualmente as subárvores AST que serão utilizadas para fazer a pesquisa na base de dados.

Já o Overcode é uma ferramenta para visualização e exploração de códigos submetidos por alunos para problemas de programação avaliados por corretores automáticos. A partir

de análises estática e dinâmica, o sistema clusteriza soluções similares automaticamente e permite que os docentes filtrem e agrupem as soluções a partir de critérios personalizados. Os resultados da pesquisa associada ao projeto mostram que ele foi capaz de diminuir o tempo necessário para professores chegarem a uma percepção geral do entendimento e dos erros dos alunos, além de proporcionar feedbacks mais relevantes para as soluções de cada estudante (GLASSMAN et al., 2015). Essa foi a ferramenta escolhida e modificada para utilização neste estudo e será abordada em mais detalhes no capítulo seguinte.

O trabalho de Huang et al. (2013) segue uma abordagem parecida, utilizando a distância de edição em árvores AST para comparar cada submissão e clusterizar as soluções. A sua limitação é que esse processo tem complexidade quadrática e exige grande poder computacional em comparação com o Overcode, que é linear (GLASSMAN et al., 2015).

3 METODOLOGIA

3.1 PLANEJAMENTO INICIAL

A metodologia foi definida considerando o objetivo deste estudo - explorar a viabilidade do agrupamento de códigos e seu uso para melhorar a confecção de feedbacks - e o principal recurso disponível para análise - a base de dados dos exercícios de programação e as respectivas respostas submetidas por alunos do Machine Teaching. Considerando todo trabalho já existente relacionado a esse tema, decidimos avaliar o agrupamento de códigos na prática, a partir da seleção de uma das ferramentas citadas no Capítulo 2 e da avaliação dos seus resultados no contexto do Machine Teaching.

Primeiro foi feita uma análise exploratória das questões e códigos obtidos do Machine Teaching, que serviu como base para a escolha da ferramenta mais apropriada, considerando também outros fatores, como a afinidade com o objetivo do estudo e facilidade de uso. Após decidir a ferramenta, foram definidas perguntas e métricas para analisar os seus efeitos em relação à avaliação de códigos e à geração de feedbacks. Por fim, foram executados todos os passos e adaptações necessárias para a obtenção desses resultados. Esse processo é descrito ao longo deste capítulo, incluindo desafios encontrados para a execução da metodologia e como foram resolvidos, além de alguns pontos importantes de implementação. O passo a passo da metodologia é resumido no fluxograma da Figura 7.

3.2 DESCRIÇÃO DA BASE DE DADOS DO MACHINE TEACHING

O Machine Teaching é compatível apenas com código escrito em Python 3. É esperado que os alunos submetam as respostas em formato de funções que já possuem o nome e a lista de argumentos pré-definidos pelo enunciado da questão, sendo possível importar bibliotecas nativas do Python. A base de dados do Machine Teaching oferece as seguintes informações:

- Enunciado das questões;
- Códigos das tentativas dos alunos, indicando se estão corretas ou não e a qual aluno cada tentativa pertence;

Figura 7 – Fluxograma da metodologia utilizada.

- Casos de teste para as funções desenvolvidas pelos alunos, compostos pelos argumentos fornecidos e os valores de retorno esperados;
- E um código correto possível para cada questão, funcionando como um gabarito.

Esta base contém 434.337 registros de tentativas de 3.162 alunos diferentes. A relação entre tentativas bem-sucedidas e mal-sucedidas é altamente desbalanceada: há 64.978 (14,96%) tentativas bem-sucedidas e 369.359 (85,04%) tentativas mal-sucedidas. Esse comportamento é esperado, pois são permitidas aos alunos quantas tentativas quanto queiram. Eventualmente, os alunos acertam a questão, ficando todas as tentativas disponíveis para análise para o professor, mas com destaque para a primeira tentativa bem-sucedida.

Também fizemos uma análise selecionando apenas a tentativa mais recente de cada aluno para cada questão. Essa seleção é importante, porque geralmente é apenas a última tentativa enviada pelo aluno que será considerada pelo professor para a avaliação. Fazendo essa filtragem, reduzimos o total de tentativas para 51.015 soluções, apenas 11,75% do montante inicial. Além disso, a situação se inverte em relação à corretude: há 45.495 (89,18%) soluções corretas e 5.520 (10,82%) incorretas. Esse comportamento também é esperado, já que os alunos podem realizar correções nos códigos a cada nova tentativa e não possuem incentivos para submeter novos códigos depois de acertarem. As análises posteriores considerarão apenas essa seleção com a última submissão de cada aluno para cada questão, ou seja, tentativas anteriores serão descartadas na contagem da quantidade de soluções para cada questão e cada aluno pode contribuir com no máximo 1 solução por questão para essa métrica.

Em relação aos exercícios de programação, existem 842 questões na base de dados. No entanto, apenas 138 possuem alguma tentativa associada e desses apenas 124 possuem códigos submetidos por 2 ou mais alunos diferentes. Nas próximas análises, apenas essas 124 questões (e suas respectivas 51.001 soluções) serão consideradas, por representarem o desafio de avaliar múltiplas submissões de alunos para exercícios de programação.

As informações acima descritas sobre a base de dados estão organizadas na Tabela 2 e na Tabela 3.

Tabela 2 – Métricas gerais da base de dados do Machine Teaching

Alunos	3.162
Soluções	434.337
Questões	842
Questões com 1 solução ou mais	138
Questões com soluções de mais de 1 aluno	124

O histograma da Figura 8 mostra a distribuição do número de soluções por cada uma das 124 questões. A média é de 411,3 soluções por questão, com desvio padrão de 554,4. Fica evidente a grande variância nesse número quando olhamos os extremos: o mínimo

Tabela 3 – Métricas de soluções corretas e incorretas

Métricas	Base de dados original	Base de dados com apenas as últimas tentativas
Total de soluções	434.337	51.015
% de soluções corretas	14,96%	89,18%
% de soluções incorretas	85,04%	10,82%
Total de soluções corretas	64.978	45.495
Total de soluções incorretas	369.359	5.520

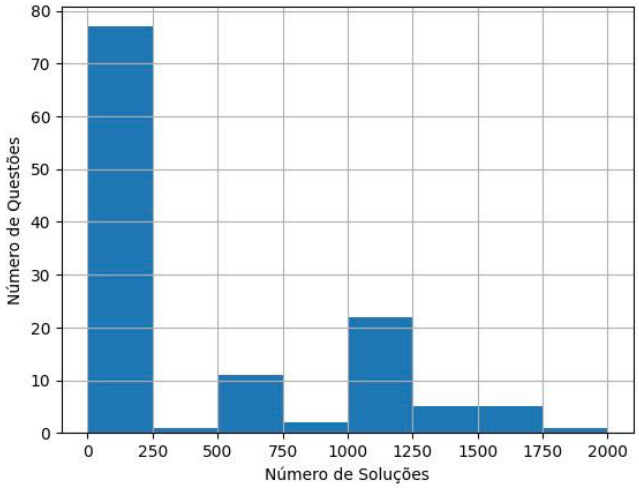

Figura 8 – Histograma do número de soluções por questão

encontrado é de 2 soluções para uma questão e o máximo é de 1813 soluções para outra questão. Além disso, o primeiro quartil é de 6 soluções, a mediana é de 23,5 soluções e o terceiro quartil é de 1013,5 soluções. Percebe-se também que a maioria das questões (77 ou 62,1%) concentra-se no bloco que possui entre 0 e 250 soluções, enquanto o resto (47 ou 37,9%) se distribui de maneira mais espalhada entre os blocos que vão de 250 a 2000 soluções. Essa distribuição sugere a partição das questões em dois tipos: as questões com "poucas" soluções ($<= 250$) e as questões com "muitas" soluções (> 250).

Já o histograma da Figura 9 mostra a distribuição do número de soluções para as questões com "poucas" soluções. Verifica-se que a quantidade de tentativas varia de 2 a 50, com média de 12,5 soluções por questão, desvio padrão de 10,7 e mediana 7. Nota-se que 45 (58,44%) das 77 questões desse tipo possuem no máximo 10 soluções.

Por fim, o histograma da Figura 10 mostra a distribuição para as questões com "muitas" soluções. O número de soluções varia de 492 a 1.813, com média de 1.064,7 soluções por questão, desvio padrão de 345 e mediana 1.065.

As quantidades de questões e soluções de cada uma das partições estão organizadas na Tabela 4. Já as estatísticas para a métrica de soluções por questão para cada uma das partições estão descritas na Tabela 5.

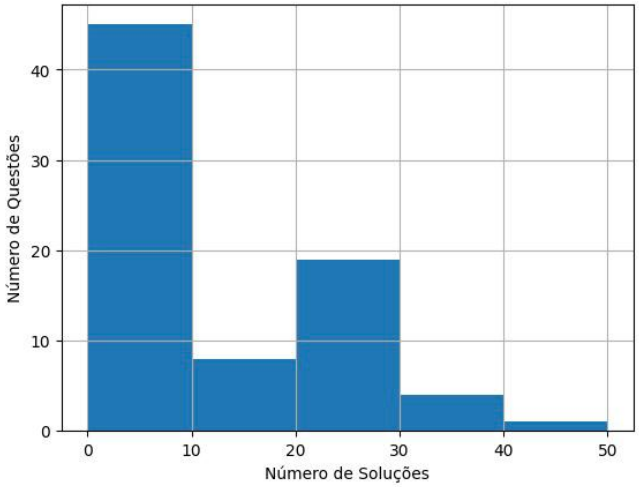

Figura 9 – Histograma do número de soluções por questão - questões com poucas soluções

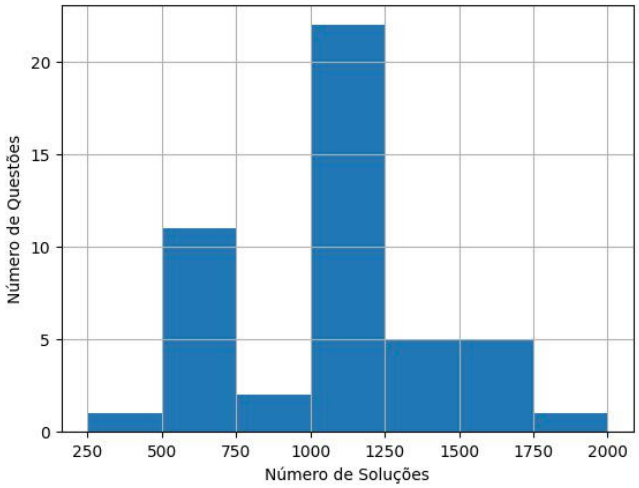

Figura 10 – Histograma do número de soluções por questão - questões com muitas soluções

Tabela 4 – Número de questões e soluções para cada partição

Métricas	**Nº de questões**	**Nº de soluções**
Todas as questões	124	51.001
Questões com "poucas" soluções	77	960
Questões com "muitas" soluções	47	50.041

3.3 ESCOLHA DA FERRAMENTA PARA O ESTUDO

Os seguintes critérios foram utilizados para a escolha da ferramenta mais adequada:

- **Alinhamento com o objetivo do estudo**: espera-se que a ferramenta já tenha sido utilizada no âmbito de melhorar a avaliação e geração de feedbacks em exercícios de programação ou em um contexto similar ao do Machine Teaching;
- **Compatibilidade da ferramenta com os dados disponíveis do Machine Teaching**: espera-se que os dados do Machine Teaching possam ser aproveitados

Tabela 5 – Estatísticas para a métrica de soluções por questão de cada partição

Partições	Média	DP	MIN	MAX	Q1	Q2	Q3
Todas as questões	411,3	554,4	2	1.813	6	23,5	1.013,5
Questões com "poucas" soluções	12,5	10,7	2	50	4	7	21
Questões com "muitas" soluções	1.064,7	345	492	1.813	824,5	1.065	1.245

sem grandes esforços para adaptação ou processamento.

- **Facilidade de uso**: espera-se que a ferramenta tenha uma utilidade clara e que possa ser rápida e facilmente adotada pelos seus potenciais usuários, os professores, sem exigir um grande esforço de adaptação.

- **Requisitos computacionais e tempo de execução**: O Machine Teaching é um projeto que conta com recursos financeiros e computacionais limitados para a operação de seu sistema. Por exemplo, não há um cluster de GPUs dedicadas ao processamento de dados do projeto. Por isso, espera-se que a ferramenta não exija muitos recursos computacionais para ser executada em um intervalo de tempo viável. De preferência, que seja possível executá-la num computador pessoal.

- **Facilidade de implementação**: a equipe de desenvolvimento do Machine Teaching é composta majoritariamente por alunos de graduação. Logo, espera-se que a ferramenta não exija conhecimentos técnicos e/ou teóricos profundos para a sua aplicação no contexto do Machine Teaching.

- **Disponibilidade do Código-Fonte**: espera-se que o código-fonte da ferramenta esteja disponível publicamente em algum repositório, sob alguma licença de software livre, permitindo o seu uso e ajuste neste projeto.

Começamos o processo eliminando aquelas alternativas que estavam mais distantes dos critérios. Primeiramente, os softwares de comparação de texto, por terem um objetivo diferente: a análise de códigos em um contexto de engenharia de software. E mesmo que fosse possível fazer uma adaptação, ainda esbarraríamos na limitação de capacidade dessas ferramentas, que analisam dois ou três arquivos por vez, enquanto o número de códigos por questão pode passar das centenas, como visto na Seção 3.2.

Por sua vez, os detectores de plágio foram eliminados porque são voltados para a área de educação, mas não com a intenção de melhorar a percepção do professor sobre o progresso dos alunos. Apesar de serem construídos para lidar com dados parecidos com os do Machine Teaching, oferecem apenas alertas contra um comportamento que se deseja evitar (o plágio), em vez de insights que possam contribuir para feedbacks mais relevantes.

As abordagens de *code embedding* também foram descartadas porque envolvem operações matriciais, que costumam exigir processamentos mais custosos e dependentes de GPUs quando envolvem grandes volumes de dados. Além disso, abordagens baseadas em Aprendizado de Máquina também costumam exigir conhecimentos aprofundados em Álgebra Linear, Estatística, Probabilidade e ferramentas de análise de dados, que podem não ter sido ensinadas aos alunos de graduação no momento em que fazem parte do projeto.

Os métodos de síntese de programas parecem boas alternativas a princípio porque seus objetivos estão bastante alinhados ao deste projeto, oferecendo funcionalidades como geração automática de feedback e clusterização de soluções. No entanto, ao observar em detalhes cada uma das ferramentas ficam claras algumas limitações:

- A solução proposta por Singh, Gulwani e Solar-Lezama (2013) exige que os professores façam uma modelagem prévia dos erros esperados em cada questão, o que dificulta sua adoção.

- O trabalho de Rivers e Koedinger (2013) depende da existência de um conjunto de códigos corretos para poder oferecer feedback aos estudantes com dificuldades. A quantidade de exemplos corretos necessários para o funcionamento do sistema não é estimada no artigo. O problema é que, conforme visto na Seção 3.2, algumas questões do Machine Teaching possuem poucas respostas ou até mesmo nenhuma. Isso é comum também no momento em que novas questões são introduzidas na plataforma.

- O artigo de Head et al. (2017) propõe uma clusterização que pode ocorrer de 2 formas: a primeira apresenta um problema similar ao do trabalho de Rivers e Koedinger (2013), porque depende da existência de submissões corretas, cuja quantidade ideal novamente não é especificada. Já o segundo caminho exige que os professores utilizem a interface do sistema para corrigir alguns códigos de alunos, o que gera uma carga de trabalho a mais para a adoção da ferramenta. Apesar desses problemas, foi a ferramenta mais promissora entre as citadas anteriormente e seu código-fonte está disponível em https://github.com/ace-lab/refazer4CSteachers.

Por fim foram analisadas as ferramentas de árvores sintáticas. O Codewebs (NGUYEN et al., 2014) exige uma montagem manual de subárvores sintáticas pelos professores para que o espaço de soluções dos alunos possa ser explorado, dificultando sua adoção. Já o trabalho de Huang et al. (2013) apresenta benefícios parecidos com os do Overcode (GLASSMAN et al., 2015), com uma análise ainda mais detalhada dos códigos de alunos, porém com complexidade quadrática em vez de linear.

Dessa forma, a última ferramenta da lista, o Overcode, se mostrou a opção mais promissora para ser avaliada neste estudo por ser a que mais atendia aos critérios de seleção:

- **Alinhamento com o objetivo do estudo**: é um software que busca tornar mais eficiente a visualização dos códigos submetidos por alunos para questões de programação.
- **Compatibilidade da ferramenta com os dados disponíveis do Machine Teaching**: foi planejado para funcionar com um formato de código similar ao utilizado nas submissões do Machine Teaching.
- **Facilidade de uso**: apresenta uma interface gráfica e interativa para visualização dos resultados, que são processados de maneira determinística a partir dos próprios códigos de alunos. Ou seja, não exige nenhum tipo de pré-processamento dos professores ou dados históricos para o seu correto funcionamento.
- **Requisitos computacionais e tempo de execução**: a complexidade é linear em relação ao número de submissões a serem analisadas e o artigo de Glassman et al. (2015) mostra que é possível completar a execução em poucos minutos, num computador pessoal, para quantidades de respostas similares às encontradas na base de dados do Machine Teaching.
- **Facilidade de implementação**: o Overcode foi implementado em Python e não exige grandes conhecimentos de teoria ou de outras ferramentas para compreendê-lo. No entanto, foi feito a partir do Python 2, uma versão da linguagem que foi descontinuada e substituída pelo Python 3. Logo, em uma nova implementação da ferramenta, será preciso adaptar seu código à versão mais recente da linguagem.
- **Disponibilidade do Código-Fonte**: o código-fonte está disponível em um repositório público no *GitHub* (https://github.com/eglassman/overcode).

No geral, apesar de sua proposta ser mais simples que as das ferramentas de *code embedding* ou síntese de programas, é justamente pela simplicidade que ela se encaixa mais facilmente em uma rotina de correção de exercícios do Machine Teaching, sem exigir grandes esforços de adaptação à ferramenta por parte dos professores. Na Tabela 6 são detalhados os desafios que foram levados em conta na comparação entre as ferramentas.

3.4 FERRAMENTA ESCOLHIDA: OVERCODE

Após a execução de um script de linha de comando que inicia o processamento dos códigos, é possível acessar a interface do Overcode por meio do navegador, em uma URL local. Nessa página podemos visualizar cada um dos clusteres onde as soluções foram agrupadas, na forma de códigos padronizados, que apresentam comportamento equivalente a de todas as soluções ali representadas. Os nomes da variáveis nessa representação são aqueles que foram mais utilizados pelos usuários, para facilitar a compreensão do código.

Tabela 6 – Desafios no uso das ferramentas para a pesquisa exploratória com o Machine Teaching

Ferramentas	Desafios
diff, meld, git diff	Detecta similaridades textuais e não equivalências sintáticas ou semânticas. Voltado para comparações individuais e não de grandes grupos.
MOSS; STRANGE	Detecta códigos similares, mas sem agrupá-los. Objetivo não é o feedback.
Code2Vec; Proto-Transformer	Operação mais custosa e dependente de GPUs (processamento paralelo). Exigem conhecimentos mais profundos de Álgebra Linear e Estatística.
Singh, Gulwani e Solar-Lezama (2013)	Exige uma modelagem prévia pelos professores para cada questão.
Rivers e Koedinger (2013)	Depende da existência prévia de códigos corretos para uma questão.
FixPropagator e MistakeBrowser	Ambas são voltadas apenas para soluções que não passaram nos testes unitários. Uma das ferramentas depende da existência de códigos corretos para uma questão. A outra exige que os professores façam correções pela própria interface para agrupar os códigos.
Codewebs	Exige que os professores montem árvores sintáticas para cada problema.
OverCode	Compatível apenas com Python 2.
Huang et al. (2013)	Complexidade computacional alta (quadrática).

Além disso é possível visualizar de maneira destacada quais linhas de código diferem cada código padronizado entre si, além da quantidade de submissões que cada um representa. Também é possível visualizar as linhas de código mais comuns nas submissões e filtrar os clusteres onde essas linhas aparecem. Por fim é possível criar regras de substituição, ou seja, definir equivalências entre linhas de código diferentes, de forma que clusteres que eram diferenciados por essas linhas passem a fazer parte de um cluster só. Por exemplo, pode ser definido que a linha "a = a + 1" é equivalente a linha "a += 1"). Ou seguindo o exemplo da Figura 11, o professor poderia criar uma equivalência entre "from math import ceil" e "from math import *", o que juntaria os 2 primeiros clusteres da segunda coluna na interface, formando um novo cluster com 31 soluções.

A padronização e agrupamento de códigos, o uso de uma nomenclatura consistente para as variáveis com as mesmas sequências de valores e a interface que destaca as diferenças entre agrupamentos de códigos têm como objetivo diminuir a carga cognitiva dos professores ao analisar uma grande quantidade de soluções. A interface também oferece a opção de marcar quais agrupamentos já foram analisados, como é demonstrado na Figura 12. Nesse exemplo, ao analisar os 5 agrupamentos com mais códigos, o professor revisou o equivalente a 72% das soluções submetidas. Sem o Overcode, para chegar à mesma

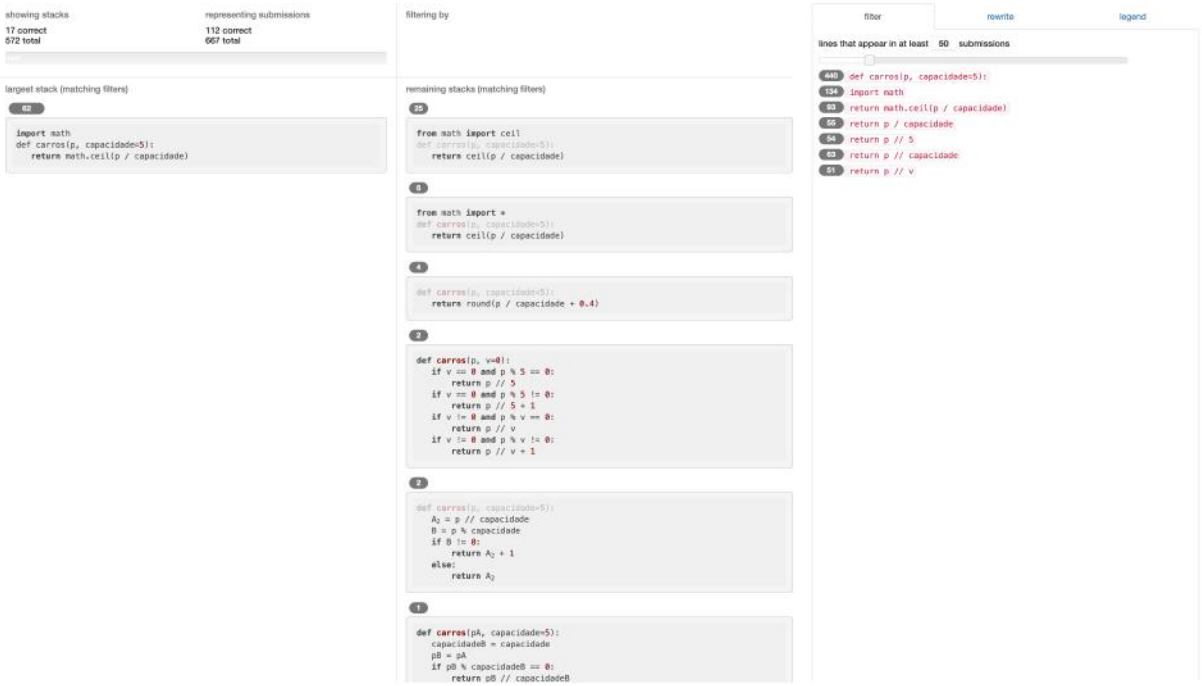

Figura 11 – Interface do Overcode com os dados de uma questão extraída do Machine Teaching.

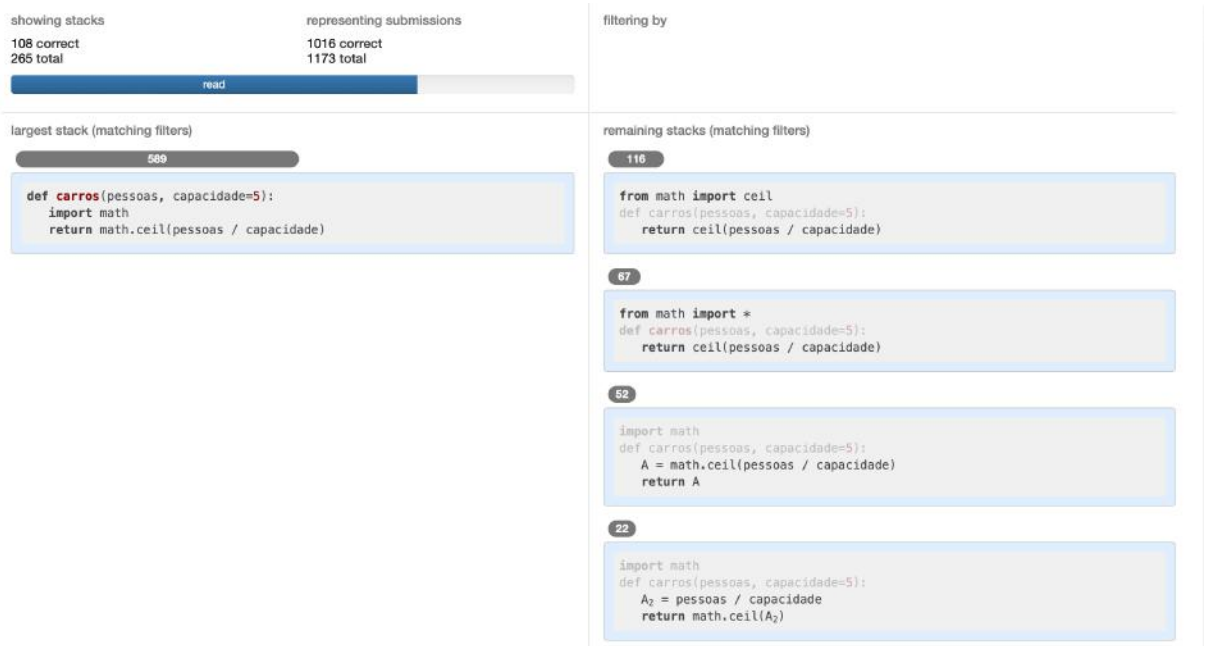

Figura 12 – Marcando as questões já analisadas na interface do Overcode.

marca, teria que analisar 846 soluções individualmente.

O artigo de Glassman et al. (2015) relata que o Overcode foi projetado para professores e monitores de cursos introdutórios de programação e propõe três usos principais para a ferramenta:

- **Identificação dos erros e lacunas mais comuns no conhecimento dos estudantes**. Por exemplo, a partir do destaque de linhas de códigos diferentes fica

mais fácil identificar soluções que usaram mais linhas em seus programas do que era necessário e qual foi a frequência desse erro.

- **Definição de Critérios de Avaliação**. Em vez de passar por cada solução e atualizar os critérios (ou rubricas) para as notas ao longo da correção, o Overcode permite que o professor tenha uma visão geral das soluções e defina critérios apropriados desde o início, evitando o retrabalho.
- **Identificação de exemplos valiosos didaticamente**. A partir do agrupamento de soluções, o Overcode facilita a percepção dos múltiplos caminhos tomados para resolver as questões, o que pode ser abordado em aula para ampliar o entendimento dos alunos sobre as possibilidades na programação.

3.4.1 Etapas de processamento do Overcode

De maneira mais detalhada, o processamento do Overcode consiste em um *pipeline* composto por 6 etapas, realizadas sobre todas as soluções submetidas por alunos para uma questão específica. Vamos abordar cada etapa, acompanhando a execução do pipeline sobre uma questão de exemplo. Será utilizada a mesma questão da Figura 11, com o seguinte 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."

A seguir, a descrição do pipeline:

1. **Reformatação**: na primeira etapa, todos os código passam por uma "limpeza" e padronização, onde comentários são deletados e espaçamentos e quebras de linha são uniformizados. Abaixo observamos o processo de reformatação da solução A, representada no Código 1, resultando no Código 2.

Código 1 – Código original da solução A

```
import math

def carros(pessoas, capacidade=5):
        ''' Define quantos veiculos de capacidade '
           capacidade' sao necessarios para carregar
            'pessoas' pessoas '''
```

```
        return math.ceil(pessoas/capacidade)
```

Código 2 – Código reformatado da solução A

```
import math
def carros(pessoas, capacidade=5):
    return math.ceil(pessoas / capacidade)
```

2. **Execução**: depois da reformatação, os códigos são executados e são feitos registros com os valores de variáveis e argumentos durante cada passo da execução. Os valores de retorno da função também são salvos. Esse processo é realizado para cada um dos casos de teste passados da questão em análise. Seguindo o exemplo, é como se a função escrita pelo aluno (e agora reformatada) fosse chamada com os valores dos casos de teste, conforme ilustrado no Código 3.

Código 3 – Execução da solução A com os dados dos casos de teste

```
carros(6)
carros(6, 7)
carros(4, 2)
```

3. **Extração de sequências de valores**: a partir dos registros salvos na etapa anterior, definem-se as sequências de valores que cada variável ou argumento de um programa assume. Exemplo: se um índice i começando em zero foi utilizado durante 4 rodadas, é guardada a sequência "i: 0, 1, 2, 3". Na questão dos carros, as sequências de valores para o primeiro caso de teste - `carros(6)` - com a solução A podem ser observadas na Tabela 7. Por ser um código relativamente simples, com 1 iteração, as variáveis assumem apenas um valor ao longo da execução.

Tabela 7 – Sequência de valores para a solução A do problema dos carros

Variáveis	Sequência de Valores
`pessoas`	6
`capacidade`	5

4. **Identificar variáveis em comum**: nesta etapa, todos os registros de programas são analisados em busca de variáveis que assumem a mesma sequência de valores, que são agrupadas em seguida. No exemplo dos carros, os Códigos 4 e 5 e as respectivas Tabelas 8 e 9 mostram que apesar de nomes diferentes, os argumentos das soluções B e C carregam a mesma sequência de valores que os argumentos `pessoas` e `capacidade` da solução A (Código 2). Esses argumentos com comportamento similar são agrupados.

Código 4 – Solução B para o problema dos carros

```
import math
def carros(p, c=5):
    return math.ceil(p / c)
```

Tabela 8 – Sequência de valores da solução B

Variáveis	Sequência de Valores
p	6
c	5

Código 5 – Solução C para o problema dos carros

```
from math import ceil
def carros(pessoas, capacidade=5):
    return ceil(pessoas / capacidade)
```

Tabela 9 – Sequência de valores da solução C

Variáveis	Sequência de Valores
pessoas	6
capacidade	5

5. **Renomear variáveis**: os nomes originais das variáveis e argumentos de cada grupo são substituídos pelo nome mais frequente do grupo, em cada um dos programas, seguindo algumas regras para evitar colisões de nomes. No exemplo dos carros, os nomes mais frequentes nos respectivos grupos de argumentos são `pessoas` e `capacidade`. O Código 6 mostra como a solução B - que utilizava nomes diferentes para seus argumentos - ficou após a etapa de renomeação. Já é possível perceber que após essa etapa, a solução B se tornou idêntica à solução A, o que será aproveitado na etapa de empilhamento de soluções.

Código 6 – Solução B após ter seus argumentos renomeados

```
import math
def carros(pessoas, capacidade=5):
    return math.ceil(pessoas / capacidade)
```

Como esse é um exemplo mais simples não foi necessário lidar com a colisão de nomes. Outra observação é que após a atualização e adaptação do Overcode para este estudo, o programa renomeia corretamente os argumentos das funções, mas apresenta um bug ainda não resolvido ao renomear variáveis, utilizando nomes genéricos como `A` e `B` em vez dos nomes mais populares de cada grupo de variáveis. Apesar de afetar a facilidade de leitura do código, esse problema não afeta o agrupamento de soluções e não afetou os resultados da pesquisa exploratória.

6. **Empilhar soluções**: depois de renomear as variáveis, muitas linhas de código se tornarão iguais e essa propriedade é aproveitada para agrupar soluções que compartilham o mesmo conjunto de linhas de código, independentemente da ordem em que aparecem na solução. Esses grupos formados são os clusteres ou pilhas que serão apresentados na interface. Além de linhas em comum, programas de um mesmo cluster devem apresentar os mesmos outputs para todos os casos de teste. Na interface sempre é escolhido um membro aleatório de cada cluster para representá-lo como código canônico. No caso das 3 soluções que estamos tratando, as soluções A e B formarão um cluster, representado no Código 7, enquanto a solução C formará outro, que aparece no Código 8. Apesar de diferentes, os 2 grupos compartilham os mesmos argumentos, valores de retorno e até uma das linhas de código.

Código 7 – Cluster com soluções A e B

```
import math
def carros(pessoas, capacidade=5):
    return math.ceil(pessoas / capacidade)
```

Código 8 – Cluster com solução C

```
from math import ceil
def carros(pessoas, capacidade=5):
    return ceil(pessoas / capacidade)
```

Se utilizássemos a funcionalidade de definir equivalência entre linhas, seria possível estabelecer a equivalência entre as duplas "`import math`" e "`from math import ceil`" e "`return math.ceil(pessoas / capacidade)`" e "`return ceil(pessoas / capacidade)`". Isso uniria as 3 soluções em um cluster só, representado no Código 9.

Código 9 – Cluster com soluções A, B e C, após a equivalência de linhas

```
import math
def carros(pessoas, capacidade=5):
    return math.ceil(pessoas / capacidade)
```

3.5 PERGUNTAS DE PESQUISA E MÉTRICAS DE AVALIAÇÃO

Para elaborar as perguntas que guiariam a investigação da ferrramenta, foram consideradas três informações importantes:

- O objetivo principal da pesquisa - explorar a viabilidade do uso de agrupamento de códigos e seu uso para aumentar a eficiência da produção de feedbacks para exercícios introdutórios de programação;

- As especificidades da base de dados disponível e das capacidades do Overcode;
- E o fato de que no artigo original do Overcode (GLASSMAN et al., 2015) a experiência do usuário já é avaliada. Em dois experimentos com 24 usuários, o Overcode mostrou-se uma alternativa mais eficiente, fácil de usar e informativa em relação a uma interface que analisava as soluções individualmente, com uma diferença estatisticamente significativa. No entanto, faltou uma análise quantitativa da capacidade do Overcode em reduzir o número de códigos a serem corrigidos pelos professores.

A partir dessas informações, foram definidas as seguintes perguntas de pesquisa:

P1 O quanto o Overcode reduz a quantidade de códigos a serem analisados pelos professores no Machine Teaching?

P2 Qual é o tempo de execução do Overcode para as questões do Machine Teaching? É um tempo que viabiliza sua utilização por professores?

Foram escolhidas as seguintes métricas para ajudar a responder as perguntas:

1. **Métricas P1**: Razão entre clusters e soluções para todas as questões e para cada grupo de questão (com poucas e muitas soluções). Essa métrica representa qual foi o resultado da redução do espaço original de soluções. Por exemplo, se temos 400 clusteres e 1.600 soluções, significa que o espaço de soluções a serem analisadas foi reduzido para 25% do tamanho original.
2. **Métricas P2**: Tempo de execução da pipeline do Overcode e o tempo geral de execução para todas as questões e para cada tipo de questão.

3.6 ATUALIZAÇÃO DO OVERCODE E INTEGRAÇÃO COM O MACHINE TEACHING

Ao longo dos testes com o Overcode, foram identificados alguns desafios para a sua integração com o Machine Teaching. Dentre eles, um dos principais é o fato de ter sido descontinuado: seu repositório oficial não é atualizado desde outubro de 2016. Isso se reflete principalmente na linguagem em que foi construído, Python 2, que não é compatível com códigos escritos na versão oficial dessa linguagem atualmente, o Python 3. Mas esse não é o único problema gerado pela falta de atualizações. Ao longo deste trabalho foi possível detectar alguns bugs e problemas na própria organização do código-fonte do Overcode, como a presença de muitos arquivos desnecessários e o acoplamento extremo entre alguns componentes, impedindo que fossem facilmente alterados.

Outro ponto que dificulta o seu uso é o seu atual formato enquanto ferramenta, já que consiste em uma aplicação de linha de comando, que funciona localmente e exige grande esforço de configuração manual por parte do usuário.

O código do projeto disponível no repositório do GitHub[1] serviu como a base para a ferramenta desenvolvida neste artigo. Originalmente, o Overcode foi desenvolvido em Python 2 e apenas analisava códigos nessa mesma linguagem. Logo, para poder utilizá-lo na análise das soluções em Python 3 submetidas por alunos no Machine Teaching foi necessário reescrever o código do Overcode também para o Python 3, já que não existe retrocompatibilidade entre as duas versões.

O processo de atualização começou pelo uso da biblioteca *2to3*[2], nativa do Python 3, que faz a conversão entre as versões de maneira automatizada. No entanto, essa solução não é perfeita, principalmente quando se trata de dependências de bibliotecas externas. Por isso, após essa conversão foi necessário fazer uma análise de como substituir alguns pacotes externos por alternativas compatíveis com o Python 3, com destaque para o módulo responsável pela quinta etapa (renomear variáveis), que precisou ser desenvolvido do zero, e para as bibliotecas que cuidavam da primeira etapa (reformatação dos códigos). Estas foram substituídas por outras alternativas externas e as informações envolvidas na decisão para essa substituição estão organizadas na Tabela 10.

Ainda sobre a reformatação, depois de testar algumas possibilidades, concluiu-se que utilizar o pacote *Black*[3] e depois o *Pyminifier*[4] (para Python 3) teve um efeito bastante similar ao uso das bibliotecas originais: *PythonTidy*[5], seguido do *Pyminifier* (para Python 2). As maiores diferenças foram no espaçamento e nos resíduos gerados (comentários e linhas em branco que são removidos por *script* depois). Também foi testado usar o *Pyminifier* (para Python 3) antes do *Black* e obteve-se resultados parecidos, porém mais legíveis. Essa foi a opção escolhida, já que não comprometeu o funcionamento do sistema. Os resultados das abordagens de reformatação são ilustrados nos Códigos 10, 11 e 12.

Código 10 – Programa antes da reformatação

```
def posLetra(frase, letra, ocorrencia):

    """RGDG"""
    i = 0
    o = 0

    while i < len(frase):
        if frase[i] == letra:
            o = o +1
            if o == ocorrencia:
                return frase.index(frase[i], i)
        i = i+ 1
    else:
```

[1] https://github.com/eglassman/overcode
[2] https://docs.python.org/3/library/2to3.html
[3] https://github.com/psf/black
[4] https://pypi.org/project/pyminifier/
[5] https://pypi.org/project/PythonTidy/

```
        return -1
```

Código 11 – Programa após a reformatação do PythonTidy + Pyminifier (Python 2)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
def posLetra(frase,letra,ocorrencia):
    i=0
    o=0
    while i<len(frase):
        if frase[i]==letra:
            o=o+1
            if o==ocorrencia:
                return frase.index(frase[i],i)
        i=i+1
    else:
        return -1
```

Código 12 – Programa após a reformatação do Pyminifier (Python 3) + Black

```
def posLetra(frase, letra, ocorrencia):
    i = 0
    o = 0
    while i < len(frase):
        if frase[i] == letra:
            o = o + 1
            if o == ocorrencia:
                return frase.index(frase[i], i)
        i = i + 1
    else:
        return -1


# Created by pyminifier (https://github.com/liftoff/
    pyminifier)
```

Em relação ao código-fonte do Overcode, foi realizada uma limpeza, excluindo arquivos que não se referiam diretamente ao objetivo principal do sistema (como notebooks Jupyter, scripts e algumas visualizações). Outra contribuição foi o registro e correção de bugs detectados com a utilização do Overcode com as questões do Machine Teaching. Além disso, uma grande parte do trabalho foi focada na criação de scripts de automação e documentação, ambos com o objetivo de facilitar a integração com o Machine Teaching e a sua utilização por outras pessoas que se interessem pelo projeto. Destacam-se a criação de scripts para explorar os dados do banco de dados do Machine Teaching e a criação de uma interface de linha de comando, na qual o professor precisa apenas indicar qual

Tabela 10 – Tabela de Comparação das Bibliotecas de Limpeza de Código

Bibliotecas	**Resumo**	**Cria padrão para tabs x espaços**	**Cria padrão para nº de espaços**	**Apaga comentários e docstrings**	**Evita 1-liners**	**Apaga linhas em branco**	**Obs**
PythonTidy	Adequar códigos à PEP 8. Para Python 2 e 3.	Sim	Sim	Apenas comentários.	Sim	Não	Adiciona comentários no início.
Pyminifier (Python 2)	Minificar e obsfucar o código. Para Python 2.	Não	Não	Sim	Não	Sim	—
Limpeza Própria do Overcode	Limpeza de código com funções do Overcode.	Não	Não	Não	Não	Não	Apaga créditos, prints e chamadas à própria função.
Black	Adequar códigos à PEP 8 e a outros padrões. Para Python 3.	Sim	Sim	Não	Sim	Não	—
Pyminifier (Python 3)	Atualiza Pyminifier para Python 3.	Não	Sim	Apenas comentários.	Não	Sim	Adiciona comentários com créditos.

o identificador da questão que deseja analisar para que todo o processamento seja feito automaticamente, como é demonstrado na Figura 13.

A última contribuição tecnológica em código foi facilitar a exploração dos dados gerados pelo pipeline do Overcode. De maneira geral, toda vez que o pipeline é executado, ele retorna um arquivo JSON contendo informações sobre os clusteres e as linhas presentes em cada solução, além de uma métrica de similaridade entre soluções erradas. Para este

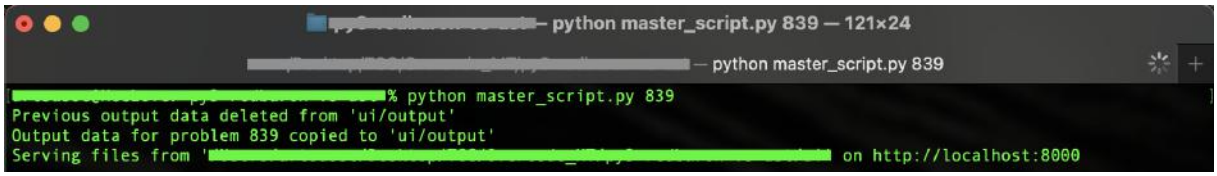

Figura 13 – Usando o Overcode pela linha de comando para processar a questão 839 do Machine Teaching.

projeto foi criado um notebook Jupyter que auxilia na análise e visualização desses dados com o objetivo de responder as perguntas de pesquisa.

O código com a ferramenta produzida neste trabalho e os notebooks de análise se encontram sob licença GPL, estando abertos e disponíveis no GitHub através do link https://github.com/artsasse/overcode/.

4 RESULTADOS

Utilizamos o Overcode, em sua nova versão desenvolvida neste trabalho, para processar uma amostra com 82 questões do Machine Teaching. Dessa amostra, 65 questões fazem parte do grupo com "poucas" soluções (até 50 soluções) e as outras 17 fazem parte do grupo com "muitas" soluções (a partir de 492 soluções).

4.1 P1: O QUANTO O OVERCODE REDUZ A QUANTIDADE DE CÓDIGOS A SEREM ANALISADOS PELOS PROFESSORES?

A Tabela 11 apresenta as estatísticas da razão entre o número de clusteres e o número de soluções para cada grupo de questões. Quando tratamos de todas as questões, o espaço de soluções a ser analisado pelos professores é reduzido, em média, para 82% do tamanho original quando o Overcode é utilizado. A análise dos quartis mostra 2 realidades: o primeiro quartil mostra que, em um quarto das questões, o espaço foi reduzido para 67% do total ou menos. No entanto, olhando o terceiro quartil, percebemos que em pelo menos 25% das questões não houve redução alguma.

Olhando apenas para as questões com poucas soluções, vemos que, em média, o espaço de soluções foi reduzido para 88% do original. Para esse grupo, a clusterização parece menos eficaz: o primeiro quartil mostra que 25% das questões reduzem seu espaço de soluções para 82% do total, enquanto o segundo quartil indica que pelo menos metade das questões não recebe benefício algum com o Overcode.

A situação muda para as questões com muitas soluções soluções. Nesse grupo, o espaço de soluções foi comprimido, em média, para 62% do original. Foi possível reduzir o número de soluções a um pouco mais da metade (55%) para o primeiro quartil de soluções e o terceiro quartil nos mostra que o número de soluções foi reduzido a 72% do original ou menos para 75% das questões. Os extremos são muito informativos também: no melhor caso, o espaço de soluções ficou 5 vezes menor (passou para 19% do original), e, mesmo no pior caso, houve uma redução significativa para 89% do original.

Tabela 11 – Estatísticas da razão entre clusteres e soluções para cada grupo de questões.

Métricas	**Média**	**DP**	**Melhor Caso (MIN)**	**Pior Caso (MAX)**	**Q1**	**Q2**	**Q3**
Clusteres/Soluções	82%	21%	19%	100%	67%	90%	100%
Clusteres/Soluções (Poucas Soluções)	88%	18%	33%	100%	82%	100%	100%
Clusteres/Soluções (Muitas Soluções)	62%	17%	19%	89%	55%	66%	72%

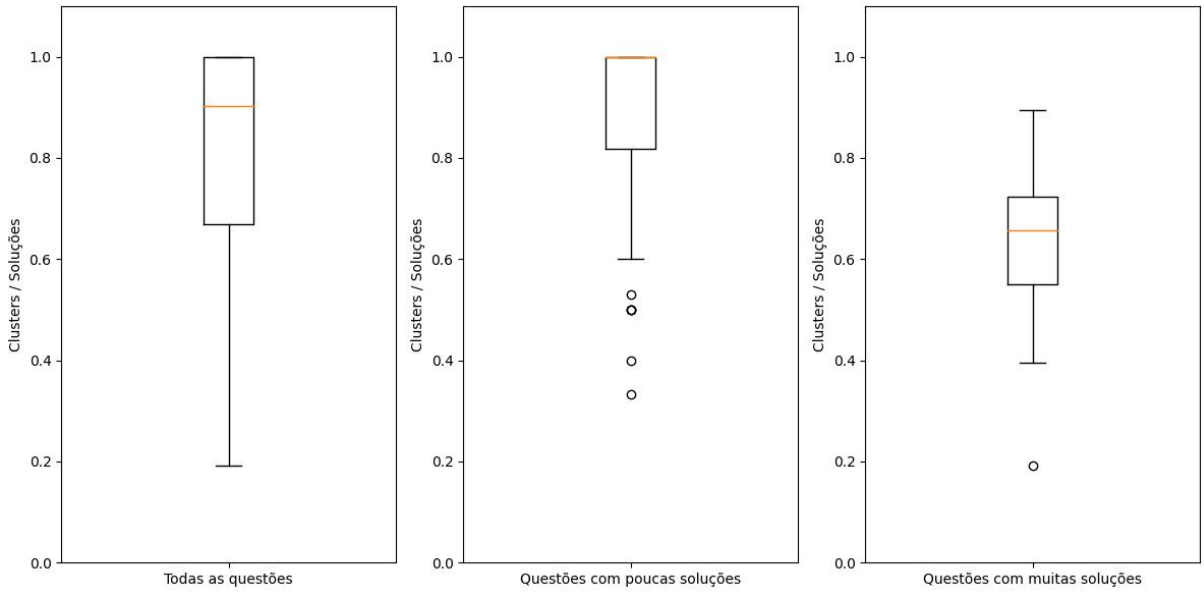

Figura 14 – Diagrama de caixa da razão entre clusters e soluções

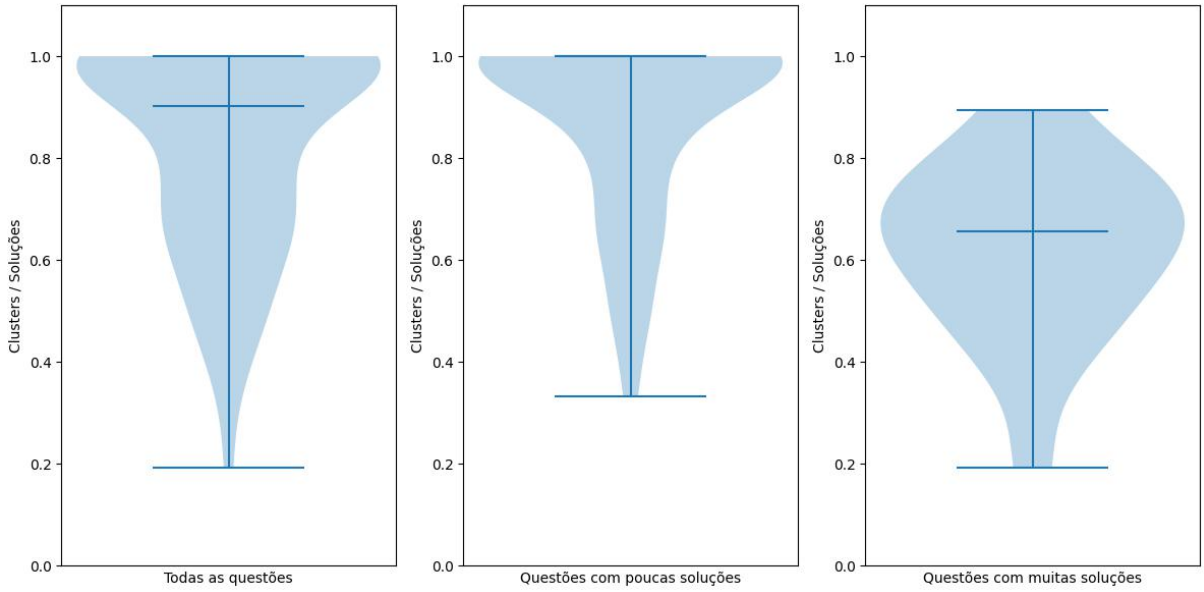

Figura 15 – Gráfico de violino da razão entre clusters e soluções

Na Figura 14 são apresentados diagramas de caixa para a razão entre clusters e soluções em cada grupo de questões. Já na Figura 15 são apresentados diagramas de violino para a mesma métrica.

Percebe-se que a redução da quantidade de códigos foi mais significativa para as questões com muitas soluções. As caixas das questões com muitas soluções e daquelas com poucas soluções não se sobrepõem no diagrama da Figura 14, sugerindo que a eficiência é diferente entre esses grupos no geral. Uma explicação possível é que quando há muitas soluções, há também uma chance maior de repetições de ideias entre alunos, que acabam sendo agrupadas nos clusteres do Overcode.

Outro fator observado que influencia a eficiência do Overcode é o número de soluções

corretas. Como o pipeline forma clusteres apenas de soluções corretas, quanto mais soluções corretas, maior a capacidade de clusterização do Overcode. Como foi mencionado na Seção 3.2, na base de dados do Machine Teaching, após selecionarmos apenas as últimas submissões de cada usuário para cada questão, obtivemos um corpo de soluções que são corretas em sua maioria (89,18%). Logo, apesar de Overcode funcionar bem para o contexto do Machine Teaching, sua capacidade de clusterização poderia ser muito menor em outro contexto, com uma porcentagem menor de soluções corretas.

4.2 P2: QUAL É O TEMPO DE EXECUÇÃO DO OVERCODE PARA AS QUESTÕES DO MACHINE TEACHING? É UM TEMPO QUE VIABILIZA SUA UTILIZAÇÃO POR PROFESSORES?

Analisamos dois intervalos de tempo: a duração do processamento das seis etapas do pipeline do Overcode e a duração total do programa, incluindo as operações de entrada e saída (incluindo a recuperação de dados da base do Machine Teaching). Foram consideradas as 82 questões da amostra, que foram processadas em um computador pessoal com as seguintes especificações: processador Intel Core i5-7267U 3.1 GHz com 2 núcleos, 8 GB RAM 2133 MHz LPDDR3, sistema operacional MacOS Ventura 13.5. A versão da linguagem Python utilizada foi a 3.10.4.

As estatísticas dos intervalos de tempo obtidos são apresentadas nas Tabela 12 e 13. Para todas as questões, o tempo médio do pipeline do Overcode foi de 2 minutos e 26 segundos e a média do tempo geral foi 3 minutos e 59 segundos. Para questões com poucas soluções as médias foram 6,1 segundos para o tempo de pipeline e 14 segundos para o tempo geral. Por sua vez, as questões com muitas soluções apresentaram tempos médios de 11 minutos e 19 segundos para o pipeline e 18 minutos e 19 segundos para a execução geral.

Os quartis dos tempos de execução do Overcode indicam que no caso das questões com poucas soluções, pelo menos 75% delas rodam em menos de 5 segundos. Já no grupo das questões com muitas soluções, pelo menos 75% rodam em menos de 9 minutos. Em ambos os grupos percebe-se uma disparidade muito grande entre o pior caso e o terceiro quartil. Detalhamos a seguir um pouco sobre essas questões fora do padrão.

A questão com poucas soluções que mais demorou para ser executada (98,3 segundos) foi a Questão 784, com o seguinte enunciado:

> Faça uma função chamada **count** que dada uma frase e uma letra, conte quantas vezes aquela letra aparece na frase, **só que sem usar a função count()**.
>
> **Atenção!** Maiúsculas e minúsculas também contam! Mas o computador não sabe que elas são iguais, você precisa antes transformá-las!

Ela recebeu apenas 2 respostas, sendo uma delas em branco e não foi possível identificar que fator levou a esse aumento inesperado no seu tempo de processamento.

Já a questão com muitas soluções que levou mais tempo no pipeline (4.937,5 segundos) foi a Questão 734, com o enunciado a seguir:

> Questão OBI (Olimpíada Brasileira de Informática - 2012, Fase 1, Nível Júnior) - (Campeonato)
>
> Dois times, Cormengo e Flaminthians, participam de um campeonato de futebol, juntamente com outros times. Cada vitória conta três pontos, cada empate um ponto. Fica melhor classificado no campeonato um time que tenha mais pontos. Em caso de empate no número de pontos, fica melhor classificado o time que tiver maior saldo de gols. Se o número de pontos e o saldo de gols forem os mesmos para os dois times então os dois times estão empatados no campeonato. Faça uma função definida por classificacao(Cv, Ce, Cs, Fv, Fe, Fs). Dados os números de vitórias e empates, e os saldos de gols dos dois times, sua tarefa é determinar qual dos dois está melhor classificado, ou se eles estão empatados no campeonato.
>
> Entrada: Os parâmetros de entrada da função são seis números inteiros C, Ce, Cs, F v, F e e F s, que são, respectivamente, o número de vitórias do Cormengo, o número de empates do Cormengo, o saldo de gols do Cormengo, o número de vitórias do Flaminthians, o número de empates do Flaminthians e o saldo de gols do Flaminthians.
>
> Saída: A sua função deve retornar a string 'Cormengo', se Cormengo estiver melhor classificado que Flaminthians ou a string 'Flaminthians' , se Flaminthians estiver melhor classificado que Cormengo; e se os dois times estão empatados a função deve retornar 'Empate'.
>
> Exemplos: Entrada: 10,5,18,11,2,18 Saída: 'Empate'
>
> Entrada: 10,5,18,11,1,18 ; Sapida: 'Cormengo'
>
> Entrada: 9,5,-1,10,2,10 Saída: 'Flaminthians'

Esse exercício recebeu 1494 respostas e não foi possível detectar nenhum fator ou anormalidade nos dados do Machine Teaching que explicasse a sua demora extra em relação as outras questões.

Nas Figuras 16, 18, 17 e 19 é possível observar diagramas de caixa e gráficos de violino para o tempo do pipeline do Overcode e para o tempo geral. Para facilitar a visualização, os dados das questões 734 e 784 foram excluídos durante a geração dessas figuras. Destaca-se a quantidade de *outliers* representados nos diagramas de caixa para todas as questões e para questões com poucas soluções. Para o primeiro grupo, isso reforça uma ideia que

já estava clara desde a análise exploratória da base de dados do Machine Teaching: a disparidade é enorme entre as questões com poucas e as questões com muitas soluções, logo é mais fácil e informativo analisar cada grupo separadamente do que tentar juntá-los em um só. Já o grande número de *outliers* entre as questões com poucas soluções indica que ainda existe uma grande diferença entre os exercícios que esse grupo concentra e para futuros estudos o ideal é buscar uma forma de dividi-lo em outros dois grupos mais uniformes. Por fim, as questões com muitas soluções apresentam um comportamento muito mais uniforme entre si.

Tabela 12 – Estatísticas do tempo de execução do Overcode para cada grupo de questões

Métricas	**Média**	**DP**	**Melhor Caso (MIN)**	**Pior Caso (MAX)**	**Q1**	**Q2**	**Q3**
Tempo (s) do Overcode	145,6	570,5	0,1	4.937,5	0,7	2,4	23,8
Tempo (s) do Overcode (Poucas Soluções)	6,1	13,6	0,1	98,3	0,6	1,2	4,8
Tempo (s) do Overcode (Muitas Soluções)	678,8	1.125,1	61,9	4.937,5	225,6	405,4	518,6

Tabela 13 – Estatísticas do tempo geral de execução para cada grupo de questões

Métricas	**Média**	**DP**	**Melhor Caso (MIN)**	**Pior Caso (MAX)**	**Q1**	**Q2**	**Q3**
Tempo (s) Geral	238,9	892,9	0,2	7.611,7	2,4	5,2	55,1
Tempo (s) Geral (Poucas Soluções)	14	22,7	0,2	133	2,1	3,7	14
Tempo (s) Geral (Muitas Soluções)	1.098,9	1.744,4	88,6	7.611,7	322,4	591,1	1.146,6

Apesar de os resultados corroborarem os relatos de Glassman et al. (2015), segundo os quais o Overcode é adequado para execução em computadores pessoais dentro de um tempo razoável, eles também mostram que o Overcode não é um programa instantâneo e que talvez seja demorado demais para contextos em que o número de soluções é muito maior do que 1.000, como em cursos abertos.

4.3 OUTRAS DESCOBERTAS DA PESQUISA EXPLORATÓRIA

Um efeito colateral positivo dos testes feitos com o Overcode em uma das questões extraídas do Machine Teaching foi a detecção de uma anomalia nos dados a partir da análise das soluções. Percebeu-se que em uma das questões, houve uma mudança no

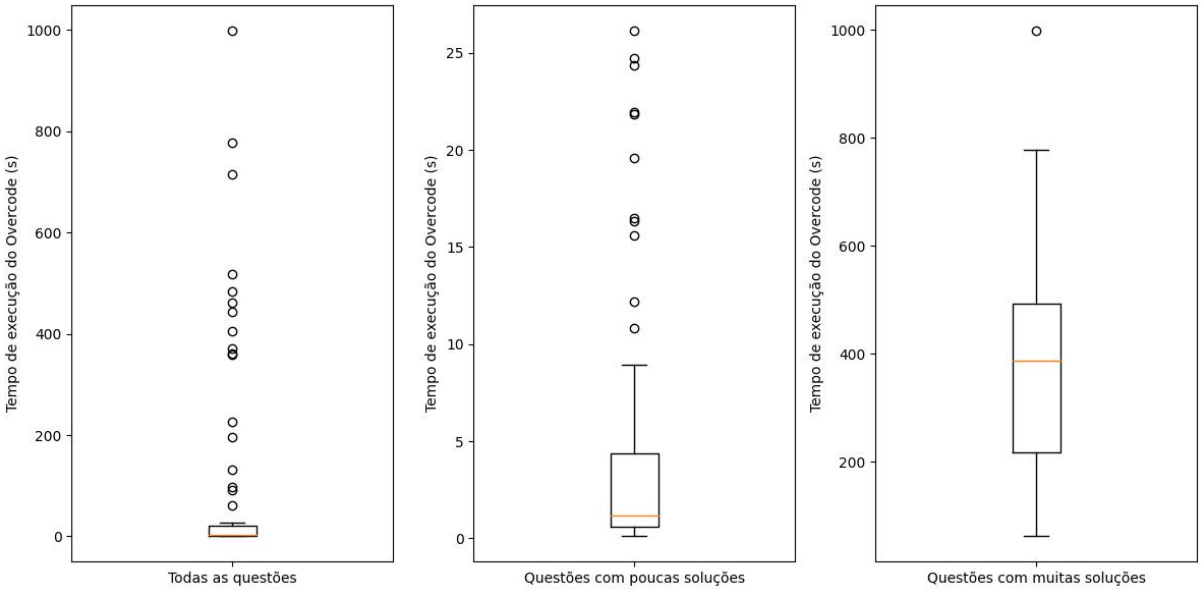

Figura 16 – Diagrama de caixa para o tempo de execução do Overcode

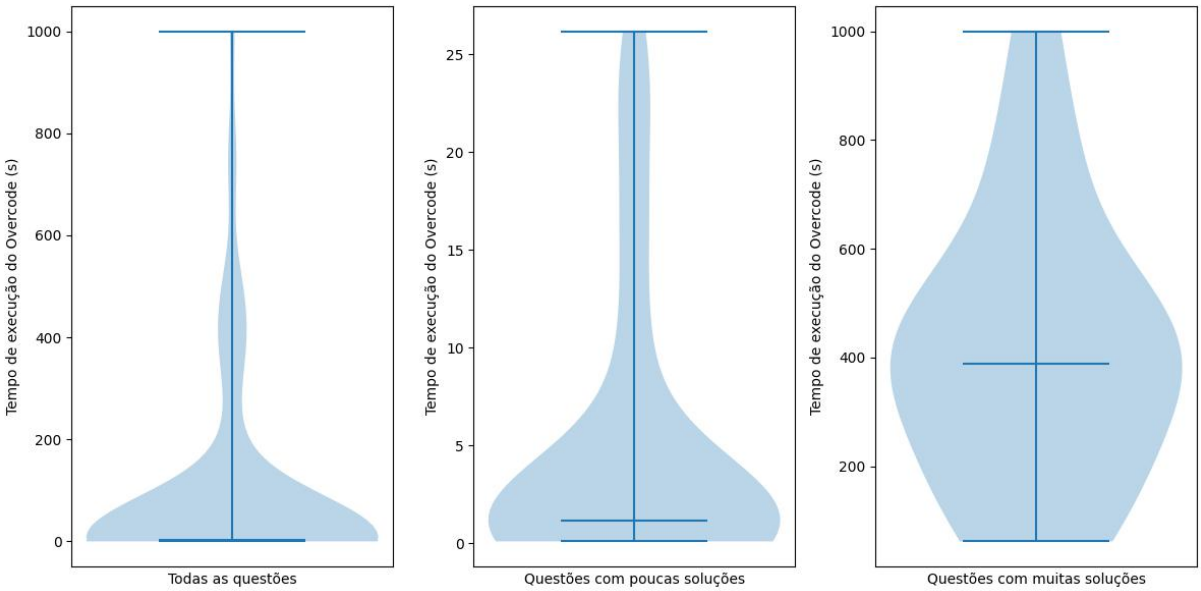

Figura 17 – Gráfico de violino para o tempo de execução do Overcode

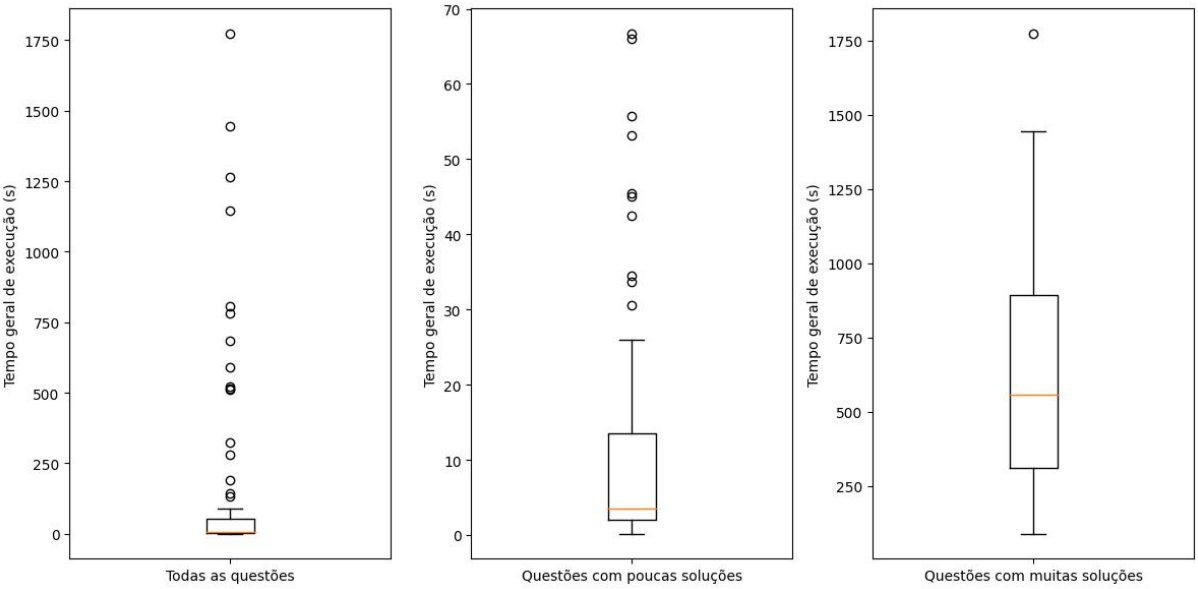

Figura 18 – Diagrama de caixa para o tempo geral de execução

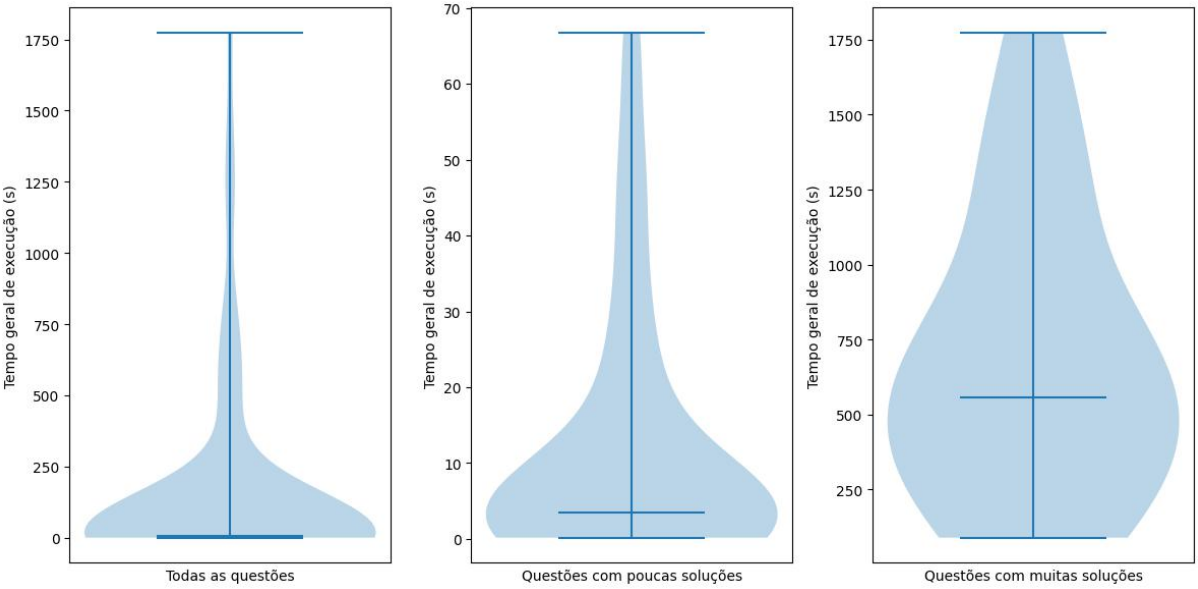

Figura 19 – Gráfico de violino para o tempo geral de execução

enunciado e no seu código de referência (talvez para adaptar a questão a uma nova turma) posterior à submissão das soluções dos alunos, que introduziu uma nova exigência na saída dos programas, tornando todas as soluções anteriores incorretas. Assim, verificou-se a necessidade de guardar as versões da questões para evitar esse tipo de problema.

Esse ocorrido sugere que o Overcode também pode funcionar como uma ferramenta de análise por parte dos administradores do sistema de ensino de programação para visualizar o comportamento das soluções e detectar comportamentos ou mesmo insights inesperados.

5 CONCLUSÃO

Partindo do objetivo de explorar a viabilidade do agrupamento de códigos como meio de aumentar a eficiência da produção de feedbacks para exercícios de programação, e, utilizando o Overcode como ferramenta aplicada à base de dados do Machine Teaching, este estudou mostrou que a abordagem de agrupar programas é capaz de reduzir o número de soluções a serem analisadas para muitas questões. Esse efeito foi verificado de maneira mais intensa nas questões com mais de 400 soluções submetidas, todas registrando algum grau de redução no número de códigos a serem analisados. Já para a amostra de questões com 50 soluções ou menos o Overcode não se mostrou tão eficaz. No entanto, isso não necessariamente indica que o Overcode não é útil em casos de turmas menores. Na verdade, muitas da questões desse grupo continham 10 ou menos soluções, o que aumenta as chances de uma maior variabilidade entre as soluções e diminui a probabilidade do Overcode encontrar padrões e agrupar respostas similares. Espera-se que em estudos posteriores seja possível confirmar a viabilidade dessa abordagem quando aplicada de maneira individual e continuada para turmas típicas de graduação, entre 20 e 100 alunos, como é o caso no Machine Teaching.

Em relação ao tempo de execução da ferramenta escolhida, o cenário se inverteu. O Overcode se mostrou relativamente rápido para o processamento de questões com poucas soluções, mas apresentou grandes gargalos e variabilidade nos tempos de processamentos para questões com muitas soluções. Isso indica que no geral, ele apresenta um tempo aceitável para um pequeno grupo de códigos, mas a partir de um certo número, esse tempo pode crescer de maneira imprevisível, talvez influenciado pelo tipo de processamento que cada questão exige. Cabe a estudos posteriores avaliar o quanto cada tipo de questão pode impactar no tempo de execução das ferramentas de agrupamento de códigos. Nossa hipótese é de que questões que exijam operações de entrada e saída ou loops com muitas iterações podem multiplicar o tempo médio necessário para completar sua execução. De toda forma, os resultados para questões com poucas soluções indicam que o Overcode ou ferramentas de complexidade similar são opções viáveis em relação ao custo computacional para o agrupamento de códigos no contexto de uma turma típica de graduação. O fato de os tempos obtidos neste estudo terem sido registrados em um computador pessoal abrem a possibilidade de utilizar o Overcode como uma ferramenta de uso local nos computadores de professores. Além disso, nossa expectativa é de que o Overcode possa obter tempos ainda melhores se executado em um servidor com hardware mais potente, tal qual o utilizado para o próprio Machine Teaching.

Além de trazer insights importantes a respeito da viabilidade e utilidade do agrupamento de códigos no ensino de programação, este estudo também culminou na reescrita do próprio Overcode, resultando em um novo software, atualizado e disponível como software

livre para uso de outros projetos e pesquisas.

Indiretamente, o estudo também trouxe insights a respeito dos dados do próprio Machine Teaching, sendo que a divisão entre questões com poucas e muitas soluções se baseou numa descoberta feita a partir da visualização das questões da plataforma em um histograma, que media o número de soluções únicas por alunos para cada questão. Enquanto o grupo de questões com poucas soluções se aproxima mais do cenário imaginado para uma turma de graduação que utiliza o Machine Teaching, o grupo com muitas soluções serve como um referencial da viabilidade do uso do Overcode nos chamados MOOCs (Massive Open Online Courses), cursos abertos, oferecidos em ambientes virtuais, que podem contar com centenas ou milhares de alunos. Apesar de ser voltado para cursos de graduação, as funcionalidades oferecidas pelo Machine Teaching são compatíveis com esses tipos de cursos abertos e os resultados obtido neste estudo facilitariam a sua potencial utilização para essa modalidade de ensino. Entre os insights específicos descobertos durante o estudo, destacam-se: o grande predomínio de soluções corretas sobre as incorretas quando consideramos apenas os últimos códigos submetidos por cada aluno e a necessidade de atenção na manutenção das questões que são reutilizadas por várias turmas no Machine Teaching ao longo dos anos, para evitar a introdução de mudanças que comprometam o histórico de corretude das questões. O potencial comprovado do Overcode para agrupar códigos similares quando há muitas soluções disponíveis também mostra-se interessante para o uso em outros estudos envolvendo o Machine Teaching que abordem a identificação de padrões a partir dos códigos, com outros objetivos além do aumento da eficiência na produção de feedbacks.

Em relação ao artigo de Glassman et al. (2015) que introduziu o Overcode, este trabalho expande os estudos sobre a ferramenta para além do aspecto da usabilidade, analisando de maneira quantitativa o seu potencial de agrupamento de soluções, aplicando o Overcode a um corpo de questões muito maior e mais variado do que aquele utilizado por Glassman et al. (2015). As novas contribuições deste trabalho deram origem a um artigo aceito no Simpósio Brasileiro de Educação em Computação (EduComp) 2024, na trilha de Recursos e Ambientes Educacionais.

Este trabalho também se diferencia do artigo de Head et al. (2017), que apesar de tratar do agrupamento de códigos, realiza experimentos com apenas 3 questões e focados na usabilidade das ferramentas que propõe, não na eficiência da clusterização de códigos. O artigo de Huang et al. (2013) trata do agrupamento de soluções que são ou funcionalmente equivalentes ou sintaticamente equivalentes, mas não necessariamente os dois como no caso do Overcode. Além disso não foi feita uma análise estatística da redução no espaço de soluções que sua abordagem permitiria. A análise feita por Nguyen et al. (2014) é a que mais se aproxima deste trabalho, no entanto é feita em cima de apenas uma questão. Além disso, não é possível uma comparação direta já que o Codewebs não é utilizado para agrupar soluções inteiras, mas para propagar feedbacks para trechos de códigos

semanticamente equivalentes. Os artigos de Rivers e Koedinger (2013), Singh, Gulwani e Solar-Lezama (2013) e Wu et al. (2019) focam na geração automática de feedbacks para soluções que não passaram por todos os testes unitários, que correspondem a apenas uma pequena parte do conjunto total de soluções consideradas neste estudo. Já os trabalhos de Alon et al. (2019), Hunt e Mcilroy (1976) e Karnalim e Simon (2021) apresentam métodos interessantes que permitem medir em algum grau a similaridade entre códigos, mas que precisam ser complementados em futuros trabalhos para que possam ser utilizados na redução do espaço de soluções.

5.1 LIMITAÇÕES E TRABALHOS FUTUROS

No geral, entendemos que os pontos importantes para trabalhos futuros se dividem em três áreas:

1. Integração com o Machine Teaching;
2. Desempenho;
3. Funcionalidades da nova versão do Overcode.

Sobre o primeiro item, como a experiência de uso da interface do Overcode e sua eficiência em reduzir o espaço de soluções já foram de alguma forma validados, falta um experimento mais amplo: integrar o Overcode à plataforma do Machine Teaching e experimentar inseri-lo na rotina de correção dos exercícios, avaliando não só o impacto para os professores, mas também a percepção dos alunos sobre a qualidade do feedback recebido. A melhor maneira de utilizar o Overcode também é uma questão aberta para futuros estudos. Uma possibilidade é aplicá-lo separadamente para cada turma ou para as turmas de cada professor, apoiando o processo de correção ao longo do período letivo. Para essa função seria necessário aprimorar sua interface e integrá-la ao Machine Teaching, permitindo o uso online. Mas também é possível usá-lo de maneira mais pontual, com foco na análise do aprendizado de todos os alunos da plataforma, comparando diferentes turmas e anos. Para isso seria necessário facilitar a comparação dos resultados do Overcode para diferentes bases de dados. Para facilitar a decisão de como o Overcode será usado, é recomendado realizar testes mais aprofundados sobre a sua dependência em relação ao número de soluções por questão. Por exemplo, a hipótese de que questões com muitas soluções apresentam uma eficiência maior pode ser validada a partir de um espaço amostral maior e por meio de um teste não-paramétrico (MONTGOMERY; RUNGER, 2010), já que não é possível assumir a normalidade dos dados do Machine Teaching.

Além disso, sobre o segundo ponto, o Machine Teaching é uma aplicação web utilizada em tempo real pelos professores durante a correção dos exercícios, portanto, é necessário reduzir o tempo de execução do Overcode. Algumas estratégias possíveis são: dividir

por turmas, diminuindo a quantidade de soluções a serem agrupadas por vez; executar o pipeline de maneira offline periodicamente, o que traria a desvantagem de não estar sempre com os clusteres atualizados; e paralelizar partes do pipeline, como a execução dos códigos de alunos. De toda forma, são caminhos que precisam de estudos mais aprofundados no futuro, além de uma investigação mais ampla sobre as causas da grande variação nos tempos de execução, mesmo para questões com muitas soluções.

Já a respeito do terceiro item, o Overcode continua sendo uma ferramenta que funciona apenas para a linguagem Python. No entanto, o pipeline descrito no artigo de Glassman et al. (2015) poderia ser implementado em outras linguagens. Também é necessário ressaltar que o Overcode, em sua origem e em sua adaptação ao Machine Teaching, é baseado em um modelo no qual os alunos submetem funções com nomes pré-determinados para serem avaliadas. Uma possível expansão de sua utilidade seria adaptar o processo do Overcode para modelos mais complexos, como a avaliação de classes ou programas inteiros. Além disso, uma lacuna do projeto original e que não foi preenchida neste trabalho é a criação de testes automatizados que cubram todo o pipeline do Overcode, aumentando a confiabilidade dos resultados.

Por fim, este estudo visou realizar uma pesquisa exploratória sobre a viabilidade do agrupamento de códigos como meio para aumentar a eficiência da produção de feedbacks no contexto de cursos introdutórios de programação. O aspecto quantitativo e de larga escala deste trabalho confirmou que o potencial do Overcode em melhorar a experiência dos professores pode ser estendido para muitas questões, com diferentes números de soluções. Apesar de seu impacto ser aparentemente maior para turmas grandes, recomendamos a sua utilização para turmas menores, como aquelas que utilizam o Machine Teaching, por consistir em uma ferramenta de baixa complexidade e fácil utilização que diminui o esforço repetitivo dos professores e permite a identificação de padrões nos códigos, contribuindo para a compreensão geral do desempenho da turma, além da produção de feedbacks.

REFERÊNCIAS

ALON, U. et al. Code2vec: Learning distributed representations of code. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 3, n. POPL, jan 2019. Disponível em: https://doi.org/10.1145/3290353.

AMBROSE, S. et al. **How Learning Works: Seven Research-Based Principles for Smart Teaching**. Wiley, 2010. (Jossey-Bass higher and adult education series). ISBN 9780470617601. Disponível em: https://books.google.com.br/books?id=gu5qpi5aFDkC.

BUTLER, M.; MORGAN, M. Learning challenges faced by novice programming students studying high level and low feedback concepts. In: ATKINSON, R. et al. (Ed.). **Proceedings of ascilite Singapore 2007 ICT: Providing Choices for Learners and Learning**. Nanyang Technological University, 2007. p. 99 – 107. ISBN 9789810595791. Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education 2007, ASCILITE 2007 ; Conference date: 02-12-2007 Through 05-12-2007. Disponível em: https://www.ascilite.org/conferences/singapore07/procs/index.html.

GIT. **Git 2.41.0 - git-diff Documentation**. 2023. [Online; acessado em 10 de Agosto de 2023]. Disponível em: https://git-scm.com/docs/git-diff.

GLASSMAN, E. L. et al. Overcode: Visualizing variation in student solutions to programming problems at scale. **ACM Trans. Comput.-Hum. Interact.**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 2, mar 2015. ISSN 1073-0516. Disponível em: https://doi.org/10.1145/2699751.

HEAD, A. et al. Writing reusable code feedback at scale with mixed-initiative program synthesis. In: **Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale**. New York, NY, USA: Association for Computing Machinery, 2017. (L@S '17), p. 89–98. ISBN 9781450344500. Disponível em: https://doi.org/10.1145/3051457.3051467.

HUANG, J. et al. Syntactic and functional variability of a million code submissions in a machine learning mooc. In: **International Conference on Artificial Intelligence in Education**. [S.l.: s.n.], 2013.

HUNT, J. W.; MCILROY, M. D. An algorithm for differential file comparison. **Computer Science**, 1976. Disponível em: http://www.cs.dartmouth.edu/%7Edoug/diff.pdf.

KARNALIM, O.; SIMON. Explanation in code similarity investigation. **IEEE Access**, v. 9, p. 59935–59948, 2021.

MONTGOMERY, D.; RUNGER, G. **Applied Statistics and Probability for Engineers**. John Wiley & Sons, 2010. ISBN 9780470053041. Disponível em: https://books.google.com.br/books?id=_f4KrEcNAfEC.

MORAES, L. et al. Machine teaching: uma ferramenta didática e de análise de dados para suporte a cursos introdutórios de programação. In: **Anais do II Simpósio Brasileiro de Educação em Computação**. Porto Alegre, RS, Brasil: SBC, 2022. p. 213–223. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/educomp/article/view/19216.

NGUYEN, A. et al. Codewebs: Scalable homework search for massive open online programming courses. In: **Proceedings of the 23rd International Conference on World Wide Web**. New York, NY, USA: Association for Computing Machinery, 2014. (WWW '14), p. 491–502. ISBN 9781450327442. Disponível em: https://doi.org/10.1145/2566486.2568023.

PRATHER, J. et al. Metacognitive difficulties faced by novice programmers in automated assessment tools. In: **Proceedings of the 2018 ACM Conference on International Computing Education Research**. New York, NY, USA: Association for Computing Machinery, 2018. (ICER '18), p. 41–50. ISBN 9781450356282. Disponível em: https://doi.org/10.1145/3230977.3230981.

PRICE, B.; PETRE, M. Teaching programming through paperless assignments: An empirical evaluation of instructor feedback. **SIGCSE Bull.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 3, p. 94–99, jun 1997. ISSN 0097-8418. Disponível em: https://doi.org/10.1145/268809.268849.

RIVERS, K.; KOEDINGER, K. Automatic generation of programming feedback; a data-driven approach. In: **International Conference on Artificial Intelligence in Education**. [S.l.: s.n.], 2013.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In: **Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2003. (SIGMOD '03), p. 76–85. ISBN 158113634X. Disponível em: https://doi.org/10.1145/872757.872770.

SINGH, R.; GULWANI, S.; SOLAR-LEZAMA, A. Automated feedback generation for introductory programming assignments. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 48, n. 6, p. 15–26, jun 2013. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/2499370.2462195.

SOLAR-LEZAMA, A. **Program Synthesis by Sketching**. Tese (Doutorado), USA, 2008. AAI3353225.

VANSCHOREN, J. **Meta-Learning: A Survey**. 2018.

WILLADSEN, K.; KENNEDY, S.; LEGOLL, V. **Meld - Visual Diff and Merge Tool (Version 3.22.0)**. 2022. [Online; acessado em 10 de Agosto de 2023]. Disponível em: https://meldmerge.org.

WU, M. et al. **ProtoTransformer: A Meta-Learning Approach to Providing Student Feedback**. arXiv, 2021. Disponível em: https://arxiv.org/abs/2107.14035.

WU, M. et al. Zero shot learning for code education: Rubric sampling with deep learning inference. In: **Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence**. AAAI Press, 2019. (AAAI'19/IAAI'19/EAAI'19). ISBN 978-1-57735-809-1. Disponível em: https://doi.org/10.1609/aaai.v33i01.3301782.