Analisando a efetividade da formação de clusters na avaliação de exercícios de programação

Arthur M. Sasse
arthurms@ic.ufrj.br
Instituto de Computação, UFRJ

Laura O. Moraes
laura@uniriotec.br
Programa de Pós-Graduação em Informática, UNIRIO

Carla A. D. M. Delgado
carla@ic.ufrj.br
Instituto de Computação, UFRJ

Carlos E. Pedreira
carlosp@centroin.com.br
Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ

RESUMO

As plataformas online de ensino de programação têm o potencial de escalar o acesso de estudantes a uma educação de qualidade. No entanto, um passo importante do processo de aprendizagem ainda é um gargalo nesses sistemas: o feedback. Este artigo considera a formação automática de clusters de códigos como uma alternativa para reduzir o trabalho docente de produção de feedback para as atividades de prática em programação. Apresentamos uma análise das abordagens possíveis e o processo de escolha de uma ferramenta para uma avaliação mais profunda. Uma vez escolhido o Overcode, o atualizamos para funcionar com o Python 3 e avaliamos a sua capacidade de formar clusters em um conjunto de dados com mais de 100 questões e dezenas de milhares de soluções. Esses dados foram extraídos do Machine Teaching, plataforma utilizada para o ensino de programação em cursos de graduação da Universidade Federal do Rio de Janeiro.

CCS CONCEPTS

• **Social and professional topics** → *Computing education*; • **Applied computing** → **Interactive learning environments**.

PALAVRAS-CHAVE

Ensino de programação, Análise de dados educacionais, Ambiente educacional

1 INTRODUÇÃO

Velocidade. Organização. Disponibilidade. Correções automáticas. São claros os benefícios das plataformas online de ensino de programação, válidos para turmas pequenas de graduação até grandes cursos abertos, como os presentes no *Coursera*. Esse suporte permite otimizar o tempo do professor e o tempo de estudo do aluno [13]. Mas o feedback personalizado é um elemento essencial do ensino que ainda depende da análise humana e não consegue ser reproduzido em larga escala nesses sistemas. No ensino de programação, exercícios práticos, como a criação de programas ou componentes de programas, são muito comuns. Além de indicar se as respostas estão corretas, o feedback didático para essas atividades deve envolver principalmente: a identificação, por meio da detecção de erros nas respostas, de conceitos mal assimilados e sugestões de melhorias diretas para o código escrito pelos alunos [2].

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, surge a demanda por uma ferramenta que torne mais eficiente esse aconselhamento para cada aluno, sem comprometer a personalização necessária para o feedback ser efetivo [3], mesmo diante de uma quantidade massiva de respostas. De maneira concreta, se um professor tem 30 alunos e a cada semana passa 5 exercícios, então, semanalmente, ele terá que 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 escreverem seus códigos, para os quais os instrutores enviam feedbacks semelhantes.

Partindo do princípio que respostas semelhantes recebem feedbacks semelhantes, se fosse possível criar clusters de códigos que compartilham um mesmo padrão, então bastaria escrever um feedback para cada padrão, que serviria para todas as respostas do cluster referente a 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 10 feedbacks em vez de 30. A partir dessa ideia, exploramos a viabilidade dessa abordagem para diminuir o trabalho repetitivo e o tempo gasto pelos professores com a produção de feedback para respostas aos exercícios de programação.

Dessa forma, este trabalho visa conduzir uma pesquisa exploratória sobre a viabilidade de formar clusters de códigos dos alunos por padrões identificados e o uso desses clusters para tornar mais eficiente a geração de feedbacks.

Primeiro, analisamos as ferramentas disponíveis para formação de clusters de códigos e produção mais eficiente de feedback, a partir de critérios pré-definidos. A ferramenta escolhida, o Overcode [5], foi adaptada para a base de dados do Machine Teaching [10], 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:

Fica permitido ao(s) autor(es) ou a terceiros a reprodução ou distribuição, em parte ou no todo, do material extraído dessa obra, de forma verbatim, adaptada ou remixada, bem como a criação ou produção a partir do conteúdo dessa obra, para fins não comerciais, desde que sejam atribuídos os devidos créditos à criação original, sob os termos da licença CC BY-NC 4.0.
EduComp'24, Abril 22-27, 2024, São Paulo, São Paulo, Brasil (On-line)
© 2024 Copyright mantido pelo(s) autor(es). Direitos de publicação licenciados à Sociedade Brasileira de Computação (SBC).

(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?

Acreditamos que nosso estudo é um passo importante para uma proposta de otimização da construção e do reaproveitamento de mensagens de feedback para exercícios, diminuindo a carga de trabalho docente sem perda na qualidade do aprendizado.

A Seção 2 aborda trabalhos relacionados. Na Seção 3 é descrita a metodologia e o desenvolvimento do trabalho. A Seção 4 apresenta os resultados e na Seção 5 fazemos considerações sobre os resultados e o futuro da pesquisa.

2 TRABALHOS RELACIONADOS

Primeiro, detalhamos o conceito de corretores automáticos e, em especial, o Machine Teaching, ponto de partida deste trabalho. Em seguida, apresentamos ferramentas ligadas à semelhança entre códigos, facilitação de feedback e exploração de grandes quantidades de soluções para problemas de programação. As ferramentas foram divididas em cinco grupos:

- 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 das ferramentas pode ser visto na Tabela 1.

2.1 Corretores automáticos e o ambiente de aprendizado Machine Teaching

Corretores automáticos são sistemas de avaliação automatizada utilizados para corrigir resoluções de exercícios submetidas por estudantes [12]. A principal vantagem oferecida por eles aos professores é reduzir o tempo necessário para realizar correções. No entanto, alguns avaliam apenas a saída dos programas e não como os conceitos ensinados foram aplicados. Por exemplo, dois programas podem ter a mesma saída e estarem corretos, mas um deles pode apresentar problemas de estilo ou de boas práticas, que só seriam detectados pela análise de um 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.

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 passa por um ou mais testes, indicando se está correta ou não.

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 [10]. Os objetivos principais do Machine Teaching são: 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. Ou seja, além de apoiar diretamente o aprendizado dos alunos, também serve para a coleta e análise de dados educacionais [10]. 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 aprovação ou falha, com a entrada fornecida, o retorno esperado e o retorno do programa submetido. O aluno pode editar o código enviado e submetê-lo quantas vezes forem necessárias, além de dispor de um espaço para escrever e executar seus próprios casos de teste. A metodologia que dá embasamento pedagógico ao Machine Teaching e sua interface podem ser vistas em [10].

2.2 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 [8]. Algumas ferramentas similares foram criadas posteriormente, como o *meld* [19], que traz uma interface gráfica 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 [4]. Essas ferramentas não consideram a similaridade sintática e semântica dos códigos, tratados como meras cadeias de caracteres. Elas foram construídas para 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 nosso propósito.

2.3 Detecção de plágio

A questão de buscar similaridade entre códigos também foi abordada por ferramentas como o MOSS [15], cujo objetivo é detectar plágio em soluções de tarefas de programação, como aquelas submetidas no Machine Teaching, mesmo quando há alguma ofuscação como variáveis com nomes diferentes. Em complemento, o STRANGE [9], além de detectar plágios, também explica, em linguagem natural (em inglês ou indonésio), as similaridades entre os trechos selecionados.

2.4 *Code embedding*

O *code2vec* [1] utiliza redes neurais para representar trechos de código como vetores. Estes são construídos a partir das árvores sintáticas abstratas de cada trecho de código, de maneira que vetores próximos representam códigos similares. Para demonstrar sua eficácia, 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 alternativas. Segundo os

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. Similaridades explicadas por 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 e classificá-los conforme 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 Mistake-Browser	Reutilizar feedbacks para códigos similares.	Formar clusters de soluções pelas correções necessárias e propagar feedbacks dentro de cada cluster.
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.	Padronizar códigos e formar clusters com base no conjunto de linhas de cada solução.
Huang *et al.* (2013)	Reutilizar feedbacks para códigos similares.	Formar clusters com base na distância de edição em árvores entre programas.

autores, os vetores capturaram relações semânticas entre os códigos, como similaridade, analogias e combinações.

Outra abordagem baseada no *code embedding*, é o ProtoTransformer [20]. O ProtoTransformer utiliza um processo de meta-aprendizagem que 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 [18]. O aprendizado é feito a partir de poucas amostras de feedback para cada problema de programação, além de informações complementares, visando gerar feedback automático para qualquer submissão futura. Ele foi aplicado com sucesso na geração de feedback para mais de 16.000 submissões de um curso de programação universitário.

2.5 Síntese de programas

A síntese de programas consiste na geração de programas a partir de especificações de alto nível [17]. Na correção de exercícios de programação, algumas pesquisas utilizaram essa abordagem para gerar feedback individualizado para cada solução submetida. Entre elas destaca-se o trabalho de Singh, Gulwani e Solar-Lezama [16], no qual o sistema, a partir de uma implementação de referência e um modelo de erros para cada questão, consegue derivar as correções mínimas para um código se tornar correto. Em uma avaliação com milhares de alunos de cursos de programação do Massachusetts Institute of Technology (MIT), esse sistema, em média, corrigiu 64% das submissões incorretas.

Já o artigo de Rivers e Koedinger [14] descreve 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, visando aferir o progresso de cada um e informar sobre os próximos passos para alcançar a corretude.

Head *et al.* [6] apresentam o *FixPropagator* e o *MistakeBrowser*, que utilizam a síntese de programas para formar clusters de soluções de estudantes pelas mudanças necessárias para torná-las corretas, com base nas correções feitas por professores e alunos.

2.6 Árvores sintáticas

O Codewebs [11] parte da análise de árvores sintáticas abstratas (AST, do inglês *abstract syntax tree*) para abordar o desafio de gerar feedbacks para milhares de submissões. O programa cria um índice para os componentes de código que integram cada submissão. Os professores podem pesquisar 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 em corretores automáticos. A partir de análises estáticas e dinâmicas, o sistema forma clusters de soluções similares automaticamente e permite que os docentes formem clusters a partir de critérios personalizados. Em dois experimentos, o Overcode diminuiu o tempo necessário para professores chegarem a uma percepção geral do entendimento e dos erros dos alunos, além de contribuir para feedbacks mais relevantes para cada solução [5]. Essa foi a ferramenta escolhida para utilização neste estudo e será abordada em mais detalhes na Seção 3.

O trabalho de Huang *et al.* [7] utiliza a distância de edição em árvores AST para comparar cada submissão e formar clusters. No entanto, esse processo tem complexidade quadrática. O Overcode, em comparação, tem complexidade linear [5].

3 METODOLOGIA

A metodologia foi definida a partir do objetivo deste estudo - explorar a viabilidade da formação de clusters de códigos e seu uso para melhorar a confecção de feedbacks - e do principal recurso para análise - a base de dados do Machine Teaching. Considerando os trabalhos relacionados, decidimos avaliar a criação de clusters de códigos, selecionando uma das ferramentas citadas na Seção 2 e analisando seus resultados no contexto do Machine Teaching.

Primeiro, fizemos uma análise exploratória das questões e códigos do Machine Teaching, que, junto a outros critérios, serviu como base para a escolha da ferramenta. Depois, definimos perguntas e métricas para analisar os efeitos da ferramenta na avaliação dos códigos de estudantes. Por fim, executamos os passos e as adaptações necessárias para obter esses resultados. Esse processo é descrito ao longo desta seção, incluindo desafios e soluções para a execução da metodologia, além de pontos importantes de implementação.

3.1 Descrição da base de dados do Machine Teaching

O Machine Teaching é compatível apenas com código em Python 3. É esperado que os alunos submetam as respostas em formato de função com nome e 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;
- Casos de teste para cada questão, compostos pelos argumentos fornecidos e os valores de retorno esperados;
- E um código correto possível para cada questão, funcionando como um gabarito.

Essa base contém 434.337 registros de tentativas de 3.162 alunos. 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%) mal-sucedidas. Isso é esperado, já que os estudantes podem enviar quantas tentativas quiserem. Eventualmente, os alunos acertam a questão, ficando todas as tentativas disponíveis para o professor analisar, mas com destaque para a primeira bem-sucedida.

Também fizemos uma análise com apenas a tentativa mais recente de cada aluno para cada questão. Essa seleção é importante, porque geralmente é a última tentativa que será considerada para a avaliação. Fazendo essa filtragem, reduzimos o total de tentativas para 51.015, 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 tentativa e não possuem incentivos para submeter novos códigos após acertarem. As análises posteriores consideram apenas a última submissão de cada aluno para cada questão, ou seja, tentativas anteriores serão descartadas na contagem de soluções para cada questão. Dessa forma, cada aluno contribui com no máximo uma solução por questão para essa métrica.

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

Tabela 3: Métricas de soluções corretas e incorretas.

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

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

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

O histograma da Figura 1 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 encontrado é de 2 soluções para uma questão e o máximo é de 1.813 para outra. Além disso, o primeiro quartil é de 6 soluções, a mediana é de 23,5 soluções e o terceiro quartil é de 1.013,5 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 entre os blocos que vão de 250 a 2.000 soluções. Essa distribuição sugere a partição das questões em dois tipos: as com "poucas" soluções (≤ 250) e as com "muitas" soluções (> 250).

Já o histograma da Figura 2 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 12,47 soluções por questão, desvio padrão de 10,67 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 3 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 estatísticas para a métrica de soluções por questão para cada partição estão na Tabela 4.

Tabela 4: Estatísticas para a métrica de soluções por questão de cada partição.

Partições	Média	DP	MIN	MAX	Q1	Q2	Q3	Nº de questões	Nº de soluções
Todas as questões	411,3	554,4	2	1.813	6	23,5	1.013,5	124	51.001
Questões com "poucas" soluções	12,5	10,7	2	50	4	7	21	77	960
Questões com "muitas" soluções	1.064,7	345	492	1.813	824,5	1.065	1.245	47	50.041

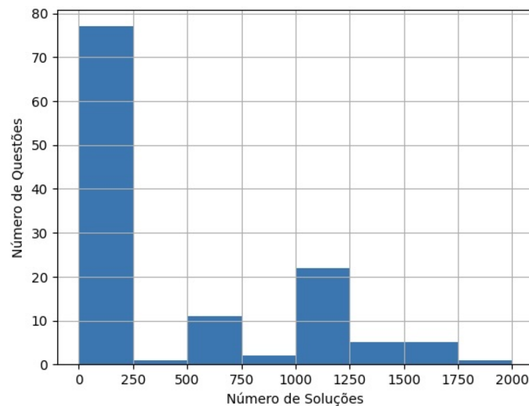

Figura 1: Histograma do número de soluções por questão.

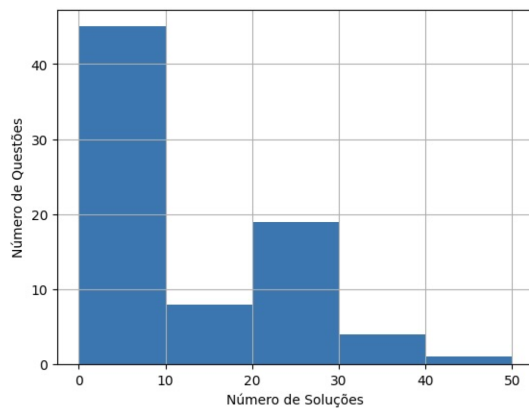

Figura 2: Histograma do número de soluções por questão - questões com poucas soluções.

3.2 Escolha da ferramenta para o estudo

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

- **Alinhamento com o objetivo do estudo**: a ferramenta deve ter 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 com os dados disponíveis**: os dados do Machine Teaching devem ser aproveitados sem grandes esforços de adaptação.
- **Facilidade de uso**: a ferramenta deve ter utilidade clara e ser de fácil adoção pelos professores, sem exigir muita adaptação.

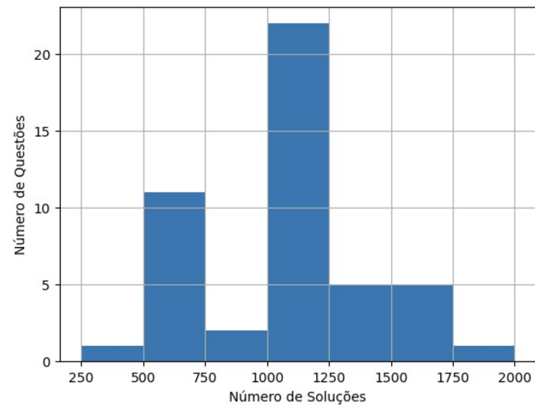

Figura 3: Histograma do número de soluções por questão - questões com muitas soluções.

- **Requisitos computacionais**: O Machine Teaching conta com recursos limitados para a operação de seu sistema. Não há um cluster de GPUs dedicadas ao projeto. Por isso, a ferramenta não deve exigir muitos recursos para ser executada em um intervalo de tempo viável. De preferência, que seja possível executá-la em um computador pessoal.
- **Facilidade de implementação**: a equipe do Machine Teaching é composta majoritariamente por alunos de graduação. Logo, a ferramenta não deve exigir conhecimentos técnicos profundos para sua aplicação.
- **Disponibilidade do Código-Fonte**: o código-fonte da ferramenta deve estar disponível publicamente, sob licença de software livre, permitindo o uso neste projeto.

Começamos o processo eliminando as alternativas 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 adaptá-los, 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 3.1.

Por sua vez, os detectores de plágio foram eliminados porque são voltados para a educação, mas não para melhorar a noção do professor sobre o progresso dos alunos. Apesar de serem construídos para dados como os do Machine Teaching, oferecem apenas alertas contra o plágio, em vez de percepções que contribuam para feedbacks mais relevantes.

As abordagens de *code embedding* foram descartadas porque lidam com operações matriciais, que costumam exigir processamentos mais custosos quando envolvem grandes volumes de dados.

Além disso, abordagens baseadas em Aprendizado de Máquina também costumam exigir conhecimentos em Álgebra Linear, Estatística, Probabilidade e ferramentas de análise de dados, que os alunos de graduação podem não ter adquirido até o 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 formação de clusters de soluções. No entanto, ao observar em detalhes cada ferramenta, ficam claras as limitações: A solução de Singh *et al.* [16] exige uma modelagem dos erros esperados em cada questão, dificultando sua adoção. O trabalho de Rivers e Koedinger [14] depende da existência de um conjunto de códigos corretos para oferecer feedback aos estudantes com dificuldades. A quantidade de exemplos necessários para o funcionamento do sistema não foi estimada. O problema é que, conforme visto na Seção 3.1, algumas questões do Machine Teaching possuem poucas respostas corretas ou até mesmo nenhuma. Isso é comum também no momento em que novas questões são introduzidas na plataforma. O artigo de Head *et al.* [6] propõe uma formação de clusters de duas formas. A primeira apresenta um problema similar ao do trabalho de Rivers e Koedinger [14], por depender da existência de submissões corretas, cuja quantidade ideal não é especificada. Já a segunda exige que os professores usem a interface do sistema para corrigir alguns códigos, gerando mais trabalho para a adoção da ferramenta. Apesar desses problemas, foi a ferramenta mais promissora entre as citadas anteriormente e seu código-fonte está disponível publicamente [1].

Por fim, foram analisadas as ferramentas de árvores sintáticas. O Codewebs [11] exige uma montagem manual de subárvores sintáticas pelos professores, dificultando sua adoção. Já o trabalho de Huang *et al.* [7] apresenta benefícios parecidos com os do Overcode [5], com uma análise ainda mais detalhada dos códigos de alunos, porém com complexidade quadrática em vez de linear.

Dessa forma, o Overcode se mostrou a opção mais promissora para este estudo por atender melhor aos critérios de seleção:

- **Alinhamento com o objetivo do estudo**: é um software que visa tornar mais eficiente a visualização dos códigos submetidos por alunos para questões de programação.
- **Compatibilidade com os dados disponíveis**: foi planejado para funcionar com um formato de código similar ao das submissões do Machine Teaching.
- **Facilidade de uso**: apresenta uma interface gráfica para visualização dos resultados, que são processados de maneira determinística a partir dos códigos dos alunos. Ou seja, não exige pré-processamento ou dados históricos para seu correto funcionamento.
- **Requisitos computacionais**: a complexidade é linear em relação ao número de submissões e o artigo de Glassman *et al.* [5] mostra que é possível executá-lo em poucos minutos, num computador pessoal, para quantidades de respostas similares às encontradas no Machine Teaching.
- **Facilidade de implementação**: não exige conhecimentos avançados para sua implementação. No entanto, foi feito em Python 2, uma versão descontinuada e substituída pelo Python 3. Logo, em uma nova implementação, será preciso adaptar seu código à versão mais nova da linguagem.
- **Disponibilidade do Código-Fonte**: o código-fonte está disponível em um repositório público no *GitHub*.

[1]https://github.com/ace-lab/refazer4CSteachers

3.3 Ferramenta escolhida: Overcode

A Figura 4 mostra a interface do Overcode. Nas colunas à esquerda e ao meio, vemos os clusters de soluções, na forma de códigos com comportamento equivalente às soluções de cada cluster. Os nomes das variáveis nesses códigos são aqueles que foram mais utilizados pelos usuários.As linhas de código que diferem cada cluster entre si são destacadas, junto à quantidade de submissões que cada cluster representa. Na coluna à direita, é possível visualizar as linhas de código mais comuns e filtrar os clusters onde elas aparecem. Por fim, é possível definir equivalências entre linhas de código, de forma que clusters que eram diferenciados por essas linhas se tornem um só. Seguindo o exemplo da Figura 4, o professor poderia criar uma equivalência entre "from math import ceil" e "from math import *", o que juntaria os dois primeiros clusters da segunda coluna, formando um novo cluster com 31 soluções. Todas essas funcionalidades visam diminuir a carga cognitiva ao analisar muitas soluções. A interface também oferece a opção de marcar quais clusters já foram analisados, conforme demonstrado na Figura 5. Nesse exemplo, ao analisar os 5 clusters com mais códigos, o professor revisou o equivalente a 72% das soluções submetidas. Sem o Overcode, para chegar à mesma marca, teria que analisar 846 soluções individualmente.

O artigo de Glassman *et al.* [5] 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**: destacando linhas de códigos que diferenciam cada cluster, é fácil identificar linhas desnecessárias e qual a frequência desse erro.
- **Definição de Critérios de Avaliação**: ter uma visão geral das soluções permite que o professor defina critérios apropriados desde o início, evitando retrabalho.
- **Identificação de exemplos didáticos**: os clusters facilitam a percepção dos múltiplos caminhos tomados para resolver as questões, o que pode ser abordado em aula.

3.4 Etapas de processamento do Overcode

O Overcode é baseado em um *pipeline* de 6 etapas, realizadas sobre todas as soluções submetidas para uma questão específica:

(1) **Reformatação**: os códigos passam por uma padronização; comentários são deletados e espaçamentos, uniformizados.
(2) **Execução**: após reformatados, os códigos são executados e os valores das variáveis registrados a cada passo da execução, assim como os valores de retorno. Esse processo é realizado para cada um dos casos de teste da questão em análise.
(3) **Extração de sequências de valores**: a partir dos registros da etapa anterior, são definidas as sequências de valores que cada variável assume durante a execução.
(4) **Identificar variáveis em comum**: variáveis que assumem a mesma sequência de valores são agrupadas.

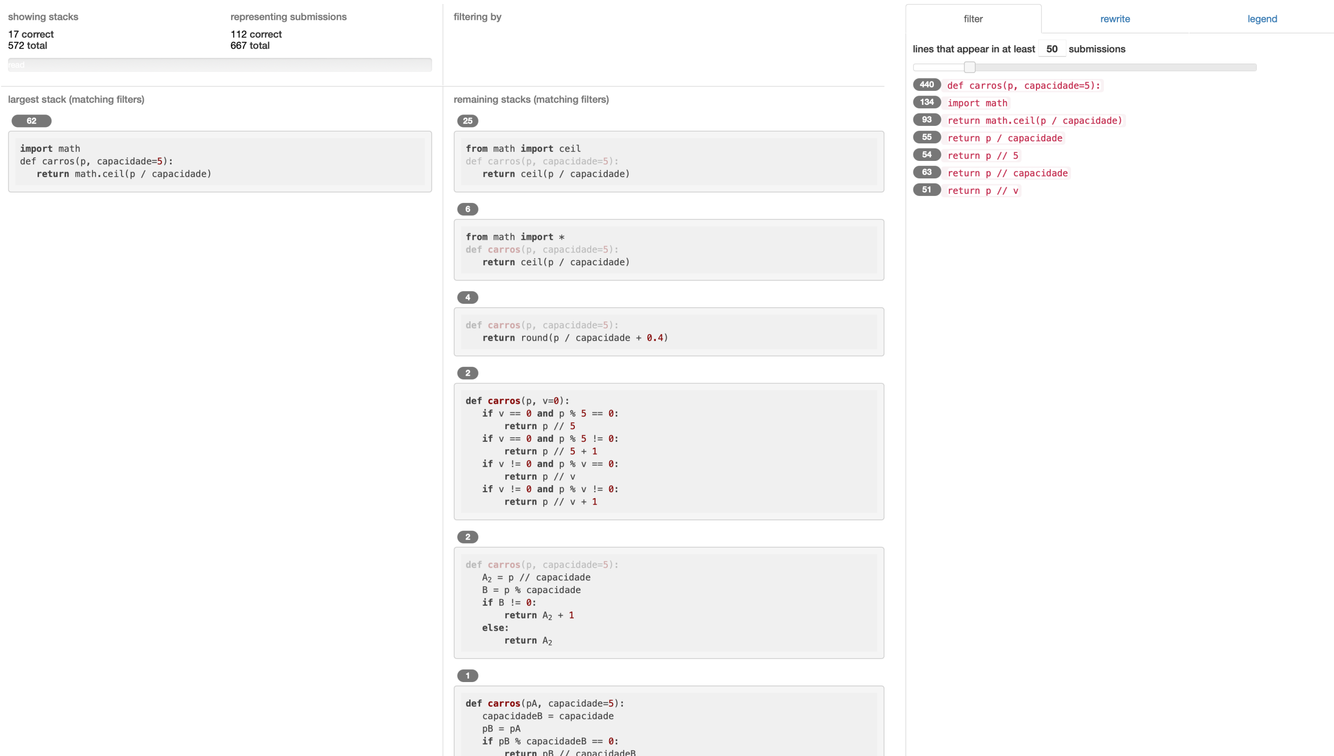

Figura 4: Interface do Overcode com os dados de uma questão extraída do Machine Teaching.

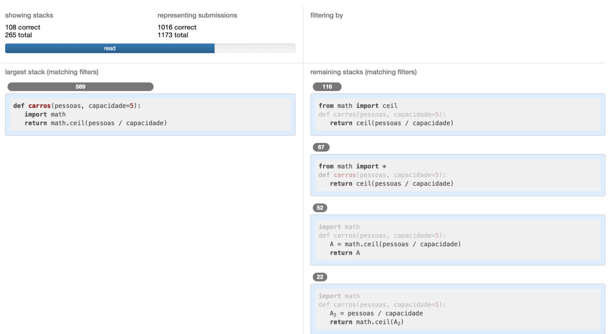

Figura 5: Marcando as questões já analisadas na interface do Overcode.

(5) **Renomear variáveis**: os nomes originais das variáveis de cada grupo são substituídos pelo nome mais frequente do grupo.

(6) **Empilhar soluções**: após renomear as variáveis, muitas linhas de código se tornarão iguais e essa propriedade é aproveitada para formar clusters de soluções que compartilham o mesmo conjunto de linhas de código, independentemente da ordem em que aparecem na solução. Esses clusters ou pilhas que serão apresentados na interface. Além de linhas em comum, programas de um mesmo cluster devem apresentar os mesmos valores de retorno para os casos de teste.

O uso e o funcionamento do Overcode são apresentados em mais detalhes no trabalho de Glassman *et al.* [5].

3.5 Perguntas de pesquisa e métricas de avaliação

Considerando nosso objetivo de reduzir o trabalho de correção de atividades de prática em programação, o artigo original do Overcode [5] deixa uma lacuna relevante. 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. No entanto, faltou uma análise quantitativa da capacidade de reduzir o número de códigos a serem corrigidos. Considerando as especificidades da base de dados disponível e das capacidades do Overcode, norteamos esta pesquisa pelas seguintes perguntas:

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 tipo 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 clusters e 1600 soluções, significa que o espaço de soluções a serem analisadas foi reduzido em 25%.

(2) **Métricas P2**: Tempo de execução da *pipeline* do Overcode 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 desafios para a sua integração com o Machine Teaching. Um dos principais é ter sido descontinuado: seu repositório não é atualizado desde 2016. Isso se reflete na linguagem em que foi construído, Python 2, que não é compatível com códigos escritos na versão oficial dessa linguagem atualmente, o Python 3. Outro problema gerado pela falta de atualizações é a presença de *bugs* e problemas na organização do código-fonte, como arquivos desnecessários e acoplamento entre componentes, dificultando alterações. Uma segunda dificuldade é o formato de aplicação de linha de comando, que funciona localmente e exige grande esforço de configuração.

O código disponível no *GitHub*[2] serviu como a base para a ferramenta desenvolvida neste artigo. Foi necessário reescrever o código do Overcode em Python 3. A atualização começou pelo uso da biblioteca *2to3*[3], nativa do Python 3, que faz a conversão entre as versões de maneira automatizada. No entanto, essa solução não é perfeita quando se trata de dependências de bibliotecas externas. Por isso, foi necessário analisar caso a caso como substituir pacotes externos por outros compatíveis com o Python 3. Os destaques foram o módulo responsável pela quinta etapa (renomear variáveis), que foi desenvolvido do zero na nova versão, e as bibliotecas da primeira etapa (reformatação dos códigos), substituídas por outras mais modernas depois de uma extensa análise.

Sobre as bibliotecas da primeira etapa, concluímos que utilizar o pacote *Black* e depois o *python-minifier* teve um efeito similar ao uso das bibliotecas originais: *PythonTidy*, seguido do *pyminifier*. As diferenças foram no espaçamento (*python-minifier* usa espaços em vez de tabulações) e nos resíduos gerados (comentários e linhas em branco que devem ser removidos depois). Também testamos usar o *python-minifier* antes do *Black* e obtivemos resultados parecidos, porém mais legíveis. Essa foi a opção escolhida, já que não comprometeu o funcionamento do sistema.

Em relação ao código-fonte do Overcode, excluímos arquivos que divergiam do foco do sistema (como *notebooks Jupyter*, *scripts* e visualizações) e corrigimos *bugs* detectados durante a utilização com dados do Machine Teaching. Outra parte do trabalho foi a criação de automações e documentação, ambos visando facilitar a integração com o Machine Teaching e a sua utilização por outras pessoas. Também criamos uma interface de linha de comando, na qual o professor fornece o identificador da questão que deseja analisar para que todo o processamento seja feito automaticamente.

A última contribuição em código foi facilitar a exploração dos dados gerados pelo Overcode. Toda vez que o *pipeline* é executado, retorna um arquivo *JSON* com informações sobre os clusters e as linhas de cada solução. Para este projeto, criamos um *notebook Jupyter* que auxilia na análise e visualização desses dados, visando responder às perguntas de pesquisa.

[2]https://github.com/eglassman/overcode
[3]https://docs.python.org/3/library/2to3.html

O código da ferramenta produzida neste trabalho e os *notebooks Jupyter* utilizados para gerar as tabelas e gráficos se encontram sob licença GPL e estão abertos e disponíveis no *GitHub* através do link https://github.com/artsasse/overcode/.

4 EXPERIMENTO E 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 são do grupo com "poucas" soluções (até 50 soluções) e as outras 17 são 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 5 apresenta as estatísticas da razão entre o número de clusters 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 revela duas 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 para as questões com poucas soluções, vemos que, em média, o espaço de soluções foi reduzido para 88% do original. Para esse grupo, a formação de clusters foi menos eficaz: o primeiro quartil mostra que 25% das questões reduzem seu espaço de soluções para 82% ou menos, 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. 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. Já 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 para 89% do original.

Na Figura 6 são apresentados diagramas de caixa para a razão entre clusters e soluções em cada grupo de questões. A redução da quantidade de códigos foi mais significativa para questões com muitas soluções. Uma explicação possível é que quando há muitas soluções, há uma chance maior de repetições de ideias, agrupadas nos clusters do Overcode.

Outro fator que influencia a eficiência do Overcode é o número de soluções corretas. Como o *pipeline* forma clusters apenas de soluções corretas, quanto mais soluções corretas, mais útil é a formação de clusters. Como foi mencionado na Seção 3.1, na base de dados do Machine Teaching, após selecionarmos apenas as últimas submissões de cada usuário para cada questão, obtivemos um corpo de soluções corretas em sua maioria (89,18%). Logo, apesar de o Overcode funcionar bem para nosso contexto, sua eficiência poderia ser muito menor em cenários com uma porcentagem menor de soluções corretas.

Tabela 5: Estatísticas da razão entre clusters e soluções para cada grupo de questões.

Métricas	Média	DP	Melhor Caso (MIN)	Pior Caso (MAX)	Q1	Q2	Q3
Clusters/Soluções	82%	21%	19%	100%	67%	90%	100%
Clusters/Soluções - poucas soluções	88%	18%	33%	100%	82%	100%	100%
Clusters/Soluções - muitas soluções	62%	17%	19%	89%	55%	66%	72%

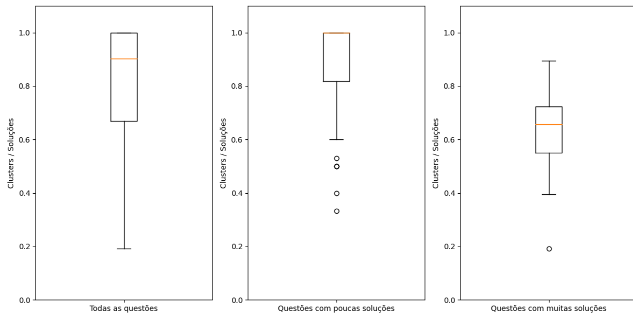

Figura 6: Diagrama de caixa da razão entre clusters e soluções.

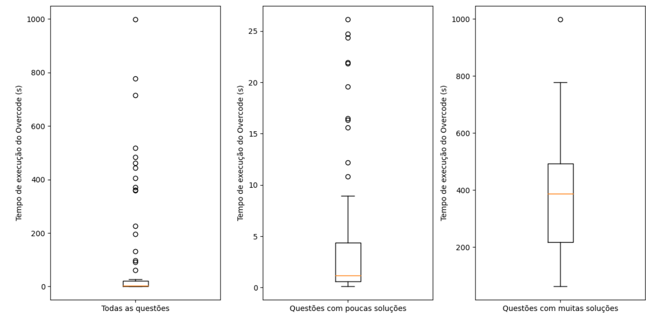

Figura 7: Diagrama de caixa para o tempo de execução do Overcode.

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 a duração do *pipeline* do Overcode, considerando as 82 questões da amostra, 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 são apresentadas na Tabela 6. Para o grupo de todas as questões, o tempo médio foi de 2 minutos e 26 segundos. Para questões com poucas soluções, a média foi 6,1 segundos. Por sua vez, as questões com muitas soluções apresentaram tempo médio de 11 minutos e 19 segundos.

Os quartis indicam que, no caso das questões com poucas soluções, pelo menos 75% delas rodam em menos de 5 segundos. Já nas questões com muitas soluções, pelo menos 75% rodam em menos de 9 minutos. Em ambos os grupos, percebe-se uma disparidade enorme entre o pior caso e o terceiro quartil.

A questão com poucas soluções que mais demorou para ser executada (98,3 segundos) foi um exercício para contar a quantidade de aparições de uma letra específica em uma frase. Ela recebeu apenas 2 respostas, sendo uma em branco. 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 um exercício para determinar, entre dois times de futebol, aquele com a melhor classificação em um campeonato a partir do número de vitórias, empates e saldo de gols de cada um. Esse exercício recebeu 1494 respostas e também não foi possível detectar nenhum fator que explicasse essa demora excessiva.

Na Figura 7 é possível observar diagramas de caixa para o tempo do *pipeline* do Overcode. Para facilitar a visualização, as duas questões fora do padrão mencionadas anteriormente foram excluídas nesses gráficos. 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 clara desde a análise exploratória dos dados do Machine Teaching: a disparidade é enorme entre questões com poucas e questões com muitas soluções, logo, é mais útil analisar cada grupo separadamente. Já o grande número de *outliers* nas questões com poucas soluções indica que existe uma grande diferença entre os exercícios que esse grupo concentra. Portanto, para futuros estudos, o ideal é buscar uma forma de dividir esse grupo em outros dois mais uniformes. Por fim, as questões com muitas soluções apresentam um comportamento mais homogêneo.

Apesar de os resultados corroborarem os relatos de Glassman *et al.* [5], segundo os quais o Overcode é adequado para execução em computadores pessoais num tempo razoável, eles também mostram que o programa não é instantâneo e talvez seja demorado demais quando o número de soluções é muito maior do que 1.000, como em cursos abertos.

5 CONSIDERAÇÕES FINAIS

Partindo do objetivo de explorar clusters de códigos como meio de aumentar a eficiência da produção de feedbacks para exercícios de programação, e, aplicando o Overcode à base de dados do Machine Teaching, mostramos que a abordagem de clusters pode reduzir significativamente o número de soluções a serem analisadas pelos professores. Esse efeito foi verificado de maneira mais intensa nas questões com mais de 400 respostas, todas registrando alguma 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, ele ainda pode ser útil para turmas menores.

Tabela 6: 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)	145,6	570,5	0,1	4.937,5	0,7	2,4	23,8
Tempo (s) - poucas soluções)	6,1	13,6	0,1	98,3	0,6	1,2	4,8
Tempo (s) - muitas soluções)	678,8	1.125,1	61,9	4.937,5	225,6	405,4	518,6

Afinal, muitas das questões desse grupo continham no máximo 10 soluções, aumentando a chance de variabilidade entre as soluções e diminuindo a probabilidade de encontrar padrões. 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.

Em relação ao tempo de execução, o cenário se inverteu. O Overcode se mostrou relativamente rápido para o processamento de questões com poucas soluções, mas apresentou grandes gargalos e variabilidade para questões com muitas soluções. Em 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. Cabe a estudos posteriores avaliar o quanto cada tipo de questão pode impactar no tempo de execução do Overcode e ferramentas similares. Nossa hipótese é que questões que exijam operações de entrada e saída ou muitas iterações podem multiplicar o tempo médio de execução. De toda forma, os resultados para questões com poucas soluções indicam que o Overcode é uma opção viável, em relação ao custo computacional, para a identificação de clusters 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 abre a possibilidade de utilizar o Overcode localmente nos computadores de professores. Também esperamos que o Overcode possa obter tempos ainda melhores se executado em um servidor mais potente, tal qual o utilizado para o próprio Machine Teaching.

Além de trazer percepções importantes sobre a utilidade dos clusters de códigos no ensino de programação, este estudo também culminou na reescrita do Overcode, resultando em um novo software, disponível como software livre para uso de outras pessoas.

Em relação ao artigo que introduziu o Overcode [5], este trabalho expande os estudos sobre a ferramenta para além da usabilidade, analisando quantitativamente o potencial de formar clusters de soluções, aplicando a um corpo de questões muito maior. Este trabalho também se diferencia do artigo de Head *et al.* [6], que apesar de tratar de clusters de códigos, realiza experimentos com apenas 3 questões e focados na usabilidade. O artigo de Huang *et al.* [7] trata da formação de clusters de soluções que são ou funcionalmente equivalentes, ou sintaticamente equivalentes, mas não necessariamente os dois, como no Overcode. Além disso, não foi feita uma análise estatística da redução no espaço de soluções. A análise feita por Nguyen *et al.* [11] é a que mais se aproxima deste trabalho, no entanto, é feita em cima de apenas uma questão, e não é possível uma comparação direta, já que o Codewebs não é utilizado para formar clusters de soluções inteiras, mas para propagar feedbacks para trechos de códigos semanticamente equivalentes. Os artigos de Rivers e Koedinger [14], Singh, Gulwani e Solar-Lezama [16] e Wu *et al.* [21] focam na geração automática de feedbacks para soluções que não passaram por todos os testes unitários, que correspondem a apenas uma pequena parte do total de soluções neste estudo. Por fim, *code2vec* [1], *diff* [8], MOSS [15] e STRANGE [9] são métodos que permitem medir a similaridade entre códigos, mas que precisam passar por grandes adaptações para serem utilizados na redução do espaço de soluções.

5.1 Limitações e Trabalhos Futuros

Como a experiência de uso da interface do Overcode e sua eficiência em reduzir o espaço de soluções já foram validados, falta um experimento mais amplo: integrar o Overcode à plataforma do Machine Teaching e inseri-lo na rotina de correção dos exercícios, avaliando o impacto para os professores e a percepção dos alunos sobre a qualidade do feedback. A maneira ideal de utilizar o Overcode também é uma questão aberta. Uma possibilidade é aplicá-lo separadamente para cada turma ou para as turmas de cada professor, apoiando a correção ao longo do período letivo. Para isso, seria necessário aprimorar sua interface e integrá-la ao Machine Teaching, permitindo o uso online. Mas também é possível usá-lo de maneira pontual, para análise do aprendizado de todos os alunos da plataforma, comparando diferentes turmas e anos. 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.

Como o Machine Teaching é uma aplicação web utilizada em tempo real durante a correção dos exercícios, é necessário reduzir o tempo de execução do Overcode. Algumas estratégias possíveis são: dividir por turmas, diminuindo a quantidade de soluções a serem analisadas por vez; executar o *pipeline* de maneira offline periodicamente, o que traria a desvantagem de não estar sempre com os clusters atualizados; e paralelizar a execução dos códigos de alunos.

Também é necessário ressaltar que no Overcode os alunos submetem funções com nomes pré-determinados para serem avaliadas. Uma possível expansão seria adaptar o seu processo para modelos mais complexos, como a avaliação de classes ou programas inteiros.

Por fim, este estudo visou realizar uma pesquisa exploratória sobre a viabilidade da formação de clusters 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 do impacto aparentemente maior em turmas grandes, recomendamos sua utilização em turmas menores, por ser uma ferramenta de baixa complexidade e fácil utilização que permite a identificação de padrões nos códigos, contribuindo para a compreensão geral do desempenho da turma, além da produção de feedbacks, com menos esforço repetitivo por parte do professor.

REFERÊNCIAS

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40, 29 pages.

[2] S.A. Ambrose, M.W. Bridges, M. DiPietro, M.C. Lovett, M.K. Norman, and R.E. Mayer. 2010. *How Learning Works: Seven Research-Based Principles for Smart Teaching*. Wiley. https://books.google.com.br/books?id=gu5qpi5aFDkC

[3] Matthew Butler and Michael Morgan. 2007. Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Proceedings of ascilite Singapore 2007 ICT: Providing Choices for Learners and Learning*, Roger Atkinson, Clare McBeath, Alan Soong Swee Kit, and Chris Cheers (Eds.). Nanyang Technological University, 99 – 107. https://www.ascilite.org/conferences/singapore07/procs/index.html 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.

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

[5] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7, 35 pages.

[6] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 89–98.

[7] Jonathan Huang, Chris Piech, An Thanh Nguyen, and Leonidas J. Guibas. 2013. Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *International Conference on Artificial Intelligence in Education*.

[8] J. W. Hunt and M. D. Mcilroy. 1976. An algorithm for differential file comparison. *Computer Science*. http://www.cs.dartmouth.edu/%7Edoug/diff.pdf

[9] Oscar Karnalim and Simon. 2021. Explanation in Code Similarity Investigation. *IEEE Access* 9, 59935–59948.

[10] Laura Moraes, Carla Delgado, João Freire, and Carlos Pedreira. 2022. 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* (Online). SBC, Porto Alegre, RS, Brasil, 213–223.

[11] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable Homework Search for Massive Open Online Programming Courses. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14)*. Association for Computing Machinery, New York, NY, USA, 491–502.

[12] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 41–50.

[13] Blaine Price and Marian Petre. 1997. Teaching Programming through Paperless Assignments: An Empirical Evaluation of Instructor Feedback. *SIGCSE Bull.* 29, 3, 94–99.

[14] Kelly Rivers and K. Koedinger. 2013. Automatic Generation of Programming Feedback; A Data-Driven Approach. In *International Conference on Artificial Intelligence in Education*.

[15] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 76–85.

[16] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *SIGPLAN Not.* 48, 6, 15–26.

[17] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

[18] Joaquin Vanschoren. 2018. Meta-Learning: A Survey. arXiv:1810.03548 [cs.LG]

[19] Kai Willadsen, Stephen Kennedy, and Vincent Legoll. 2022. Meld - Visual Diff and Merge Tool (Version 3.22.0). https://meldmerge.org [Online; acessado em 10 de Agosto de 2023].

[20] Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. ProtoTransformer: A Meta-Learning Approach to Providing Student Feedback.

[21] Mike Wu, Milan Mosse, Noah Goodman, and Chris Piech. 2019. 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* (Honolulu, Hawaii, USA) *(AAAI'19/IAAI'19/EAAI'19)*. AAAI Press, Article 97, 9 pages.