Passar nos casos de teste é suficiente? Identificação e análise de problemas de compreensão em códigos corretos

Eryck Pedro da Silva, Ricardo Edgard Caceffo, Rodolfo Azevedo
{eryck.silva,caceffo,rodolfo}@ic.unicamp.br
Universidade Estadual de Campinas
Campinas, São Paulo, Brasil

RESUMO

O uso de sistemas de correção automática (*autograders*) auxilia o ensino de disciplinas de introdução à programação (CS1). No entanto, o foco na corretude pode ofuscar a verificação de outros problemas presentes no código. Neste trabalho, foi investigado se códigos, ditos corretos por um *autograder*, apresentavam comportamentos que poderiam indicar falhas na aprendizagem dos conceitos abordados em CS1. Esses comportamentos foram denominados Problemas de Compreensão em Códigos Corretos (PC³). Ao analisar 2441 códigos, uma lista com 45 PC³ foi elaborada e posteriormente avaliada por docentes de CS1 para identificar quais PC³ mais necessitam de correção em sala de aula e de que forma essa correção poderia ser realizada. Ao todo, 15 PC³ foram considerados mais graves e as sugestões dos docentes envolveram detecção automática dos PC³ e utilização de técnicas de Aprendizagem Ativa. Os resultados obtidos podem orientar a construção de artefatos para intervenções que abordem PC³ em CS1.

CCS CONCEPTS

• **Social and professional topics → CS1**.

PALAVRAS-CHAVE

Introdução à programação, Problemas de compreensão, Avaliação automática, CS1.

1 INTRODUÇÃO

Sistemas de correção automática de código (*autograders*) costumam ser utilizados em disciplinas de introdução à programação (CS1) para auxiliar o instrutor. Em 1960, Hollingsworth [19] mencionou que o uso de *autograders* economiza tempo e dinheiro, além de permitir aumentar o número de discentes nas turmas. Atualmente, turmas que possuem muitos alunos, incluindo, principalmente, os Cursos Online Abertos e Massivos (do inglês *Massive Open Online Course*), empregam *autograders* pela dificuldade de comunicação individual dos instrutores com alunos [27].

No contexto educacional, *autograders* auxiliam nas correções de tarefas, poupando recursos e retirando parte da carga de trabalho dos instrutores [15, 16, 19]. No entanto, a utilização de *autograders* pode gerar dependências. Baniassad et al. [4] mencionam que os alunos podem se viciar em utilizar o *feedback* gerado pela avaliação automática para construírem suas soluções por tentativa e erro. Os discentes também podem sentir frustrações com a ferramenta por eventuais mal funcionamentos [22] e por uma confiança superestimada de que *autograders* não cometem erros [20].

Uma utilização comum de *autograders* no ensino de CS1 é para a verificação da saída esperada de um código [30]. Essa utilização promove estudos que visam entender e auxiliar o ensino e a aprendizagem com base em características funcionais [3, 6, 26, 28]. No entanto, em CS1, mesmo códigos corretos podem possuir características indesejáveis que programadores mais experientes não desenvolveriam [13], como, por exemplo, uma maior complexidade resultante da utilização redundante de construtos sintáticos (como demasiados desvios condicionais ou aninhamento de laços) [21, 33, 40]. Em especial, o foco na corretude pode estimular a não preocupação com fatores de legibilidade e manutenibilidade, essenciais para futuros programadores [13, 23]. Ao considerar que pode não haver uma preocupação, tanto do aluno quanto do instrutor, em analisar códigos que já geram a saída esperada, essas ou outras características que indicam falhas nos conceitos abordados em CS1 não seriam detectadas e corrigidas.

Estabelecida a motivação de uma análise de códigos corretos por um *autograder*, este trabalho teve o objetivo de identificar se esses códigos apresentam comportamentos que podem indicar falhas na aprendizagem dos conceitos abordados em CS1. A análise considerou esses comportamentos como problemas de compreensão (*misconceptions*) [31]. Em CS1, o estudo desses problemas identifica e classifica erros sintáticos, semânticos ou lógicos, cometidos pelos discentes [2, 8, 10, 17], porém, eles não se limitam a ocorrências apenas em códigos corretos. Dessa forma, o presente trabalho, ao delimitar seu estudo aplicando essa condição sobre os códigos, estabeleceu um subgrupo denominado Problemas de Compreensão em Códigos Corretos (PC³).

Considerando como hipótese a existência de PC³ em códigos corretamente avaliados por um *autograder*, o presente trabalho buscou responder as seguintes perguntas de pesquisa:

P1: Quais PC³ apresentam um grau de gravidade maior, possuindo maior prioridade de correção durante o ensino de CS1?

P2: De quais formas os PC³ têm potencial para serem corrigidos em sala de aula, considerando os contextos de ensino e aprendizagem de CS1?

Ao todo, foram identificados 45 PC³ em uma análise exploratória de códigos de uma disciplina de CS1. Ao consultar docentes que lecionam essas disciplinas, os PC³ foram ranqueados de acordo com sua gravidade e prioridade de correção em sala de aula. Dentre esses, 15 foram destacados, sendo relacionados aos conceitos de expressões Booleanas e iteração, em sua maioria, seguidos por organização de

EduComp'23, Abril 24-29, 2023, Recife, Pernambuco, Brasil (On-line)

código, uso de variáveis e funções e sobre características dos casos de teste. Os docentes também expressaram opiniões favoráveis à utilização de *autograders* que identifiquem PC³ automaticamente e sobre a aplicação de técnicas de Aprendizagem Ativa [7] para corrigir os PC³ em sala de aula, embora características como tempo de configuração e aplicação, bem como a quantidade de alunos por turma tenham sido fatores de preocupação. O *feedback* dos docentes, também composto por sugestões, pode servir para a construção de artefatos que possam ser utilizados em CS1 de modo a corrigir os PC³.

Este artigo está dividido da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; a Seção 3 descreve a metodologia empregada na realização desta pesquisa; os resultados obtidos estão descritos na Seção 4; seguidos da discussão na Seção 5. Por fim, a Seção 6 apresenta as conclusões e propostas para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Caceffo et al. [10] identificaram, por meio de análise de provas e entrevistas com instrutores de uma disciplina de CS1 que leciona a linguagem C, problemas de compreensão classificados nas seguintes categorias: uso de parâmetros de função e escopo; variáveis, identificadores e escopo; recursão; iteração; estruturas; ponteiros; e expressões Booleanas. Os autores utilizaram esses resultados para criar um Inventário Conceitual, que é um questionário de múltipla-escolha utilizado para identificar problemas de compreensão [1].

Gama et al. [17] analisaram se os problemas de compreensão identificados por [10] poderiam ser aplicados à linguagem Python. Os autores utilizaram a frequência desses problemas nas provas para mantê-los ou descartá-los na criação do novo catálogo em Python. Categorias foram desconsideradas por serem irrelevantes na linguagem (estruturas e ponteiros) e uma nova surgiu: uso e implementação de classes e objetos. O catálogo levantou 28 hipóteses de problemas de compreensão, necessitando validação futura.

Araújo et al. [2] realizaram um mapeamento empírico de problemas de compreensão ao expandir os resultados obtidos por [17] uma vez que os contextos das disciplinas de CS1 estudadas eram parecidos. Por meio da análise de avaliações parciais, os autores listaram 27 problemas de compreensão em Python, dos quais 19 estavam presentes em [17]. Os 8 restantes compuseram uma nova categoria, denominada *Adicional*, que engloba erros lógicos e sintáticos simples.

Sobre o uso de *autograders* em CS1, Araujo et al. [3] desenvolveram a *Python Enhanced Error Feedback - PEEF*: um ambiente de desenvolvimento *online* que possui mensagens aprimoradas de erro do compilador, *chat* integrado e análise dinâmica do código por meio de testes unitários. Os autores discutiram sobre o uso da ferramenta para alunos e professores de CS1. Liu e Petersen [26] criaram a PyTA, que realiza análise estática de código [42] para promover avisos e mensagens de erros de uma forma efetiva e simples de entender. Os autores aplicaram um estudo no qual alunos poderiam consultar, ou não, o *feedback* da ferramenta. Para os alunos que utilizaram a PyTA, a conclusão foi que, em relação aos exercícios propostos na disciplina, houve redução no número de erros por exercício, no total de submissões necessárias até que um erro presente fosse corrigido e no total de submissões até a solução passar nos casos de teste.

Em relação a pesquisas derivadas de análise de códigos corretos em CS1, De Ruvo et al. [13] introduziram o conceito de estilos semânticos (*semantic styles*): indicadores que podem ser manifestações de um conhecimento precário a respeito de conceitos de programação. Ao analisar submissões de discentes para listas de exercícios, os autores identificaram 16 indicadores de estilos semânticos: 12 deles estão relacionados com comandos condicionais (e.g., *else* desnecessário, código duplicado em *if/else*) e os 4 restantes abordam uso de variáveis.

Ureel e Wallace [40], motivados em promover um ciclo de *feedback* formativo próximo do instrutor, desenvolveram o WebTA. Os autores classificam a ferramenta como um crítico de código, que detecta código anômalo proveniente de construções que podem estar ou não corretas sintaticamente. Além de detectarem problemas de compreensão já mapeados (aproximadamente 200), o WebTA também oferece suporte para a criação de novas regras de detecção, que podem ser elaboradas pelo instrutor com base no que ele espera das soluções dos discentes para os exercícios de programação.

Por fim, Keuning et al. [24] construíram um Sistema Tutor Inteligente [41] que promove dicas, passo a passo, para reescrever um código correto, mas com baixa qualidade. Para a construção do sistema, os autores consultaram professores experientes, regras de aprimoramento de código presentes em ferramentas consolidadas no mercado e outras fontes na literatura. Os autores listam 19 regras presentes no sistema, envolvendo expressões Booleanas, ramificações, laços e declarações.

A Tabela 1 mostra uma comparação desta pesquisa com os trabalhos relacionados apresentados nesta seção. De uma forma geral, o presente estudo se destaca por limitar a busca de problemas de compreensão apenas em códigos corretos por um *autograder* enquanto alguns relacionados não se limitaram a essa condição [2, 3, 10, 17, 26]. Por outro lado, este trabalho se propôs a estudar dificuldades presentes em códigos corretos em CS1 na linguagem Python, diferenciando-se dos demais aplicados em outras linguagens [13, 24, 40].

Tabela 1: Comparação desta pesquisa com os trabalhos relacionados.

Pesquisa	Erros ou Problemas Abordados	Análise de Códigos Corretos	Linguagem
Caceffo et al. [10]	15	-	C
Gama et al. [17]	28	-	Python
Araújo et al. [2]	21	-	Python
Araujo et al. [3]	*	-	Python
Liu e Petersen [26]	*	-	Python
De Ruvo et al. [13]	16	✓	Java
Ureel e Wallace [40]	~200	✓	Java
Keuning et al. [24]	19	✓	Java
Esta Pesquisa	45	✓	Python

*Total não informado. Utilizam mensagens de erro aprimoradas de compilador para erros do Python.

3 METODOLOGIA

Nesta seção, é apresentada a metodologia aplicada para o estudo sobre os PC³. É descrito, inicialmente, processo de identificação, informando o contexto da disciplina de CS1 utilizada (3.1). Em sequência, é apresentado o processo de classificação da gravidade dos PC³ obtidos na lista inicial (3.2) e, por fim, é informado como ocorreu o processo de identificação e avaliação de ideias para intervenções em CS1 abordando os PC³ (3.3). A Figura 1 resume as etapas da metodologia.

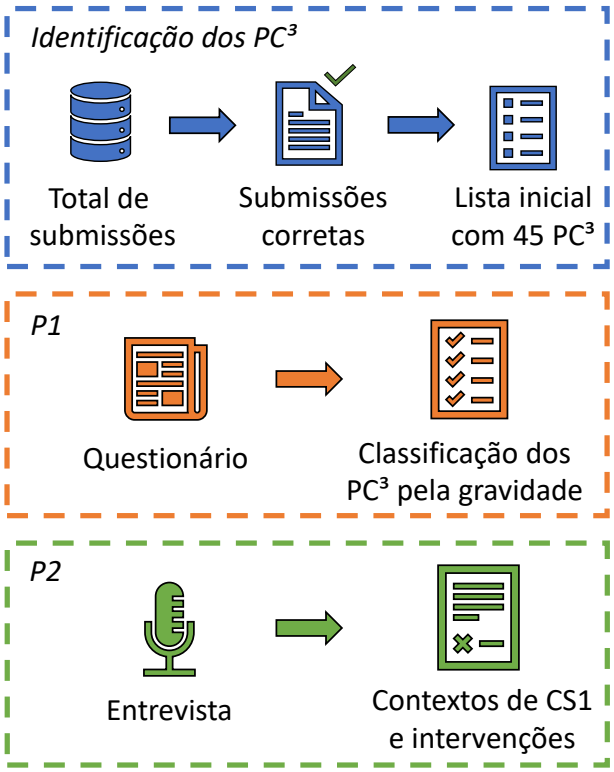

Figura 1: Descrição geral da metodologia utilizada.

3.1 Identificação dos PC³

O principal objetivo desta etapa foi identificar se códigos considerados corretos por um *autograder* apresentam características indesejáveis em relação aos objetivos de aprendizagem de disciplinas de CS1. Para isto, a disciplina Algoritmos e Programação de Computadores (MC102), ofertada na UNICAMP, foi escolhida. A disciplina possui um número elevado de alunos por semestre (cerca de 600). Além disso, ela é dividida em um grupo de turmas coordenadas: a mesma ementa é administrada para todas as engenharias e demais cursos da área de exatas da universidade. No entanto, os cursos de Engenharia da Computação e Ciência da Computação, embora sigam a mesma ementa, não fazem parte das turmas coordenadas. O paradigma de programação imperativo é ensinado na disciplina, com a utilização da linguagem Python para todas as turmas, coordenadas ou não, desde 2018.

Durante o semestre, os alunos de MC102 realizam tarefas práticas somativas, denominadas *laboratórios*. Os discentes submetem esses laboratórios por meio de um sistema de correção automática, desenvolvido pela própria instituição, o SuSy[1]. O SuSy realiza uma análise dinâmica dos códigos, com o principal objetivo de verificar se os dados de saída das soluções dos alunos são realmente os esperados para aquela tarefa. Essa verificação é feita por meio de casos de teste: conjuntos de dados que, para uma entrada, possuem uma saída pré-estabelecida. Os casos de teste podem ser abertos ou fechados: os abertos são disponibilizados para os alunos, de forma a auxiliar os discentes na construção de suas soluções; já os fechados não são. A nota do aluno é definida pela quantidade de testes nos quais suas submissões obtém sucesso. No entanto, o SuSy também é configurável para restringir número máximo de submissões (geralmente limitado a 20 para evitar que ele seja utilizado como compilador e promover acertos por tentativa e erro), limitar o tempo máximo de execução e realizar identificação de plágio.

Para identificar quais são os PC³ presentes nos códigos corretamente avaliados por um *autograder*, foi realizado um processo de coleta e análise de soluções produzidas pelos alunos para as tarefas práticas da referida disciplina.

3.1.1 Processo de Coleta. Os códigos submetidos para os laboratórios foram obtidos pelo SuSy. Ao todo, 19 turmas coordenadas da disciplina, que ocorreram no primeiro semestre letivo de 2020, tiveram seus códigos coletados. Destaca-se que todos os procedimentos de identificação dos PC³ foram realizados após as turmas já terem sido encerradas.

Como a premissa era de observar características indesejáveis, em relação aos objetivos de aprendizagem, em códigos que passavam em todos os casos de teste, uma filtragem precisou ser realizada antes da análise iniciar. Para cada envio dos discentes, o SuSy gera um arquivo de relatório contendo informações, dentre elas, o número de casos de teste que a submissão passou ou não. O sistema mantém apenas o último envio do discente, que pode ser considerado, teoricamente, a versão mais correta da solução. Utilizando esses dados foi possível filtrar quais submissões passaram em todos os casos de teste.

No período letivo mencionado, a disciplina teve um total de 15 laboratórios. Como a presente pesquisa se encontrava em processos exploratórios iniciais, foi decidido somente coletar laboratórios que foram atribuídos na primeira metade da disciplina, ou seja, antes da primeira avaliação parcial. Essa decisão levou em consideração a identificação de comportamentos indesejáveis desenvolvidos com a aprendizagem de assuntos mais básicos de CS1, indicando que, caso não sejam abordados no início, podem se manifestar nos assuntos mais complexos ensinados posteriormente. Além disso, como alguns laboratórios que ocorreram nesse intervalo abordavam o mesmo assunto, eles foram desconsiderados. Ao todo, 6 laboratórios foram escolhidos para análise, descritos na Seção 4.

3.1.2 Processo de Análise. O conjunto de códigos foi analisado de forma manual, seguindo a ordem crescente da numeração dos laboratórios. Planilhas foram construídas para organizar as observações percebidas em cada uma das submissões dos alunos. Inicialmente,

[1] https://www.ic.unicamp.br/~susy/

essas observações foram classificadas sem uma nomenclatura específica, apenas descrevendo os comportamentos observados (que se tornariam os PC³), e eventuais implicações das ocorrências. Com o avanço das análises, comportamentos similares foram percebidos e nomes provisórios foram estabelecidos a eles (atualizando os comportamentos correspondentes anteriores, se necessário), para facilitar a identificação de todos no final.

Uma vez que os códigos do conjunto selecionado foram analisados, um processo de categorização teve início, finalmente consolidando os PC³ por meio dos comportamentos observados. Para isto, as planilhas foram analisadas de forma conjunta, com o objetivo de agrupar todas as observações similares. A nomenclatura das categorias para os PC³ foi elaborada com baseada no trabalho de Gama et al. [17]. Ao todo, 2441 códigos foram analisados, gerando uma lista inicial com 45 PC³ divididos em 8 categorias (detalhes apresentados na Seção 4).

Embora seja possível afirmar que a identificação dos PC³ tenha sido abrangente por analisar 19 turmas de uma mesma disciplina de CS1, os PC³ detectados podem estar sujeitos à localidade, pois as turmas foram de uma mesma instituição de ensino; e também sujeitos à interpretação do pesquisador. De forma a mitigar essas ameaças à validade, uma consulta com docentes de CS1 foi elaborada para avaliar os PC³. Essa consulta foi composta por um questionário eletrônico, com o objetivo de classificar a gravidade de cada PC³; e uma entrevista semiestruturada [25], com a finalidade de identificar os contextos de ensino de CS1 e quais intervenções envolvendo os PC³ podem ser elaboradas. Todos os procedimentos foram realizados em inglês e de forma *online*, tanto para alcançar um público maior como pela questão da pandemia de *Sars-Cov-2*. Além disso, como as avaliações com docentes de CS1 envolviam seres humanos, a pesquisa foi avaliada e aprovada por um Comitê de Ética em Pesquisa (CEP)[2].

3.2 Classificação da Gravidade

O objetivo desta etapa foi verificar como professores de CS1 classificariam os 45 PC³ em relação à gravidade desses comportamentos. Por gravidade entende-se uma maior necessidade de explicação em sala de aula sobre esses comportamentos, já que eles estariam representando potenciais falhas na compreensão dos assuntos abordados. Além de mitigar as ameaças à validade comentadas, outro resultado esperado foi o de criar um ranqueamento dos PC³. Dessa forma, os mais graves poderiam ser identificados, tornando-se os mais propícios para investigações futuras, incluindo a elaboração de intervenções em sala de aula.

3.2.1 Processo de Coleta. A coleta de dados foi realizada por meio de um questionário eletrônico. O período de convocação ocorreu entre janeiro e fevereiro de 2022. Foram enviados convites para listas de discussão e diretamente a autores de publicações com foco em CS1. O tempo de preenchimento estimado do questionário foi de 40 a 55 minutos e foram aceitas respostas até o final de março de 2022. O roteiro foi composto pelos seguintes passos:

(1) Apresentação do Termo de Consentimento Livre e Esclarecido, descrevendo a pesquisa e pedindo o consentimento do voluntário.
(2) Questões básicas de contextualização sobre o participante: nome, instituição onde leciona, tempo de experiência no ensino de CS1, se tem experiência ensinando Python e quais outras linguagens de programação o participante tem familiaridade.
(3) Questões de classificação da gravidade dos PC³. Para cada item, foi apresentado o nome, a descrição, um exemplo genérico de código e a descrição do exemplo genérico. Duas perguntas compuseram a classificação: uma utilizando item Likert sobre a opinião do respondente em relação à afirmação de que o PC³ é grave; e outra sendo um campo de texto no qual o respondente poderia comentar demais observações sobre o referido PC³.
(4) Convocação do respondente para a entrevista semiestruturada.

3.2.2 Processo de Análise. O ranqueamento dos PC³ pela gravidade foi construído com base nas frequências obtidas das respostas dos itens Likert. Inicialmente, as frequências de grupos similares foram agrupadas (discordo totalmente (DT) e discordo (D), neutro (N) e branco (B), concordo totalmente (CT) e concordo (C)) e depois utilizadas para computar a diferença entre aqueles que concordavam que o PC³ podia ser considerado como grave e aqueles que não concordavam. Em outras palavras, essa diferença, nomeada como DIF, pode ser interpretada como $(CT + C) - (N + B + DT + D)$.

A análise dos comentários feitos pelos participantes foi conduzida utilizando análise de conteúdo [32]. Foram identificadas quatro categorias de características a respeito dos PC³: gravidade, frequência, causas da ocorrência e formas de mitigação. Embora o resultado dessa análise não tenha sido utilizado diretamente no ranqueamento dos PC³, as informações obtidas ofereceram maiores reflexões a respeito desses comportamentos, possibilitando uma compreensão maior para ações posteriores.

3.3 Intervenções para o Ensino de CS1

As entrevistas semiestruturadas foram idealizadas como convite adicional aos respondentes do questionário. Os objetivos desta etapa foram: obter informações a respeito de outros contextos de ensino e aprendizagem de CS1; descobrir se os PC³ ocorrem em outras disciplinas de CS1 e como os professores lidam com eles; e obter a opinião dos professores sobre possíveis intervenções no ensino e na aprendizagem de CS1 que dizem respeito aos PC³, buscando reduzir as ocorrências desses comportamentos.

3.3.1 Processo de Coleta. As entrevistas foram elaboradas no formato semiestruturado porque o conjunto pré-estabelecido de perguntas não se manteve fixo, devido ao caráter exploratório da pesquisa. O tempo estimado para a realização foi de 40 minutos. As entrevistas ocorreram entre março e junho de 2022 e foram conduzidas pela plataforma *Google Meet*. O roteiro geral foi composto pelos seguintes passos:

(1) Introdução sobre o pesquisador e o propósito da pesquisa. Solicitação para gravar a entrevista.
(2) Perguntas sobre o contexto de ensino de CS1, no qual, o entrevistado atua.
(3) Perguntas sobre os PC³ e como o entrevistado lida com eles.

[2]CAAE: 51444121.5.0000.5404

(4) Perguntas sobre possíveis intervenções no ensino e na aprendizagem de CS1 que abordam os PC³, e como o entrevistado as utilizaria em sala de aula.

3.3.2 Processo de Análise. Uma vez concluídas as entrevistas, as respostas para cada pergunta foram compiladas individualmente. A análise de conteúdo [32] foi aplicada, agrupando as seguintes informações: contexto de ensino, no qual, os docentes atuam (estrutura das aulas, tipos de tarefas somativas e uso de *autograders*); opiniões dos docentes a respeito dos PC³ (se já observaram, como lidam com eles e exemplos de outros similares não mencionados); e opiniões sobre possíveis artefatos para abordagem dos PC³ (utilização de *autograder* que detectasse PC³ e utilização de técnicas de Aprendizagem Ativa em sala de aula envolvendo os PC³).

4 RESULTADOS

Os resultados obtidos com as etapas da pesquisa são apresentados na ordem de execução. A identificação dos PC³ é descrita informando sobre os laboratórios analisados, bem como as categorias elaboradas para os PC³. Depois, os dados obtidos com o questionário são informados, exibindo a lista de PC³ ranqueada pela gravidade e exemplos dos considerados mais graves. Por fim, são descritos os dados obtidos com as entrevistas semiestruturadas. A versão completa dos resultados pode ser encontrada em nosso relatório técnico (em Inglês) [35].

4.1 Identificação dos PC³

Como mencionado na Seção 3, um total de 6 atividades de laboratório foram selecionadas para a análise. Ao todo, 2959 submissões foram enviadas pelos alunos da disciplina sendo que apenas 2874 estavam corretas, ou seja, passaram em todos os casos de teste. A Tabela 2 detalha os conteúdos abordados e os totais de submissões gerais, corretas e analisadas de cada laboratório. Destaca-se que foram apenas analisadas 220 (aproximadamente metade) submissões corretas dos dois últimos laboratórios porque eles abordaram um mesmo conceito (laços), com a utilização de estruturas diferentes (*while* e *for*, respectivamente).

Tabela 2: Descrição das atividades de laboratório e os totais das submetidas, corretas e analisadas da disciplina de CS1 verificada para compor a identificação dos PC³.

Conteúdo Abordado	Submissões	Corretas	Analisadas
Tipo *int*: operações aritméticas	535	529	529
Tipo *float*: operações aritméticas	499	491	491
Tipo *bool*: operações lógicas	511	503	503
Comandos condicionais	499	478	478
Laços simples: comando *while*	459	452	220
Laços aninhados: comando *for*	456	421	220
Total	2959	2874	**2441**

Com a análise das 2441 submissões corretas, foram identificados 45 PC³, divididos em 8 categorias. A lista completa está presente junto à classificação da gravidade (Tabela 3). As categorias receberam uma identificação de A até H, identificadas da seguinte forma:

- A) Variáveis, identificadores e escopo (8 PC³ - de A1 a A8)
- B) Expressões Booleanas (12 PC³ - de B1 a B12)
- C) Iteração (8 PC³ - de C1 a C8)
- D) Parâmetros de função e escopo (4 PC³ - de D1 a D4)
- E) Raciocínio (2 PC³ - E1 e E2)
- F) Casos de teste (2 PC³ - F1 e F2)
- G) Organização de código (6 PC³ - de G1 a G6)
- H) Outros (3 PC³ - de H1 a H3)

4.2 Questionário

Um total de 32 voluntários responderam ao questionário. A distribuição de respondentes por país foi a seguinte: Brasil (18), Estados Unidos da América (9), Austrália (1), Colômbia (1), Eslovênia (1), Finlândia (1) e Países Baixos (1). Nenhuma das respostas enviadas foi descartada por ser classificada como inválida, porém, alguns respondentes deixaram respostas em branco. Para a elaboração do ranqueamento dos PC³, as respostas deixadas em branco foram considerados válidas e agrupadas com os neutros na análise dos itens Likert.

A Tabela 3 apresenta o ranqueamento dos PC³ considerados mais graves. São apresentados o ID e o nome de cada PC³ em conjunto com os agrupamentos das frequências similares (concordo totalmente e concordo (CT+C), neutro e branco (N+B), discordo totalmente e discordo (DT+D)) e o cálculo da DIF (descrito na Seção 3.2). O nome de cada PC³ foi elaborado ao resumir qual é o problema de compreensão em si. Um corte, representado por uma linha horizontal, foi realizado para destacar os considerados mais graves: ao selecionar os PC³ com DIF sendo estritamente maior que 10, os 15 primeiros foram agrupados nesse quesito.

Neste trabalho, optou-se por discutir apenas os 15 PC³ considerados mais graves. Para ilustrá-los, quatro exemplos foram elaborados, cada um contendo um conjunto desses comportamentos listados na Tabela 3. É importante notar que esses exemplos foram construídos de forma genérica, não possuindo outra finalidade.

4.2.1 Código 1. Esse exemplo contém 5 PC³: A4, D4, G4, G5 e H4. A redefinição de *built-in* (A4) ocorre na linha 9, com o *built-in* **max** sendo redefinido pelo usuário como uma nova função. Nessa mesma função também ocorre acesso a variáveis fora do escopo (D4), sendo **a** e **b** declaradas nas linhas 7 e 8, respectivamente. Variáveis sem nome significativo (G4) estão presentes pelo código, como **a**, **b**, **c**, **x** e **y**. O código apresenta uma organização arbitrária de declarações (G5), pois alterna entre *input* (linha 1) com declaração de função (linha 2), seguidos de mais *input* (linhas 7 e 8) com outra declaração de função (linha 9), só então com o restante do código (linhas 14, 15 e 16). Por fim, uma declaração sem efeito (H2) está presente na linha 4, pois o resultado da função *round* não está sendo atribuído à nenhuma outra variável.

Código 1: Exemplos dos PC³: A4, D4, G4, G5 e H1.

```
1 a = int(input())
2 def foo(a):
3     a = a / 3.5
4     round(a, 2)
5     return a
6
7 b = int(input())
8 c = int(input())
```

```
9  def max():
10     if b >= c:
11         return b
12     return c
13
14 x = foo(a)
15 y = max()
16 print(x, y)
```

4.2.2 Código 2. Esse exemplo contém 4 PC³: B6, B8, B9 e B12. Um *while* está sendo utilizado desnecessariamente (B6) para checar se a soma de **num1** com **num2** é maior que 9 na linha 4, pois um *break* é utilizado logo em sequência na linha 6. Já a não utilização de *elif/else* (B8) na linha 10 pode fazer com que o valor de **res** (linhas 9 ou 11) seja sobrescrito dependendo do valor de **num2**. O *elif* declarado na linha 15 está checando, desnecessariamente, o inverso da verificação já feita no *if* da linha 13 a respeito da variável **num1** (B9). Por fim, a mesma condição é testada com *if's* duas vezes nas linhas 18 e 20 (B12), com operações distintas como resultado (linhas 19 e 21, respectivamente): essas poderiam ser agrupadas dentro de um único teste.

Código 2: Exemplos dos PC³: B6, B8, B9 e B12.

```
1  num1 = int(input())
2  num2 = int(input())
3
4  while num1 + num2 > 9:
5      print(num1 + num2, "tem mais de 1
           digito")
6      break
7
8  if num2 <= 0:
9      res = num1 * num2
10 if num2 % 2 == 0:
11     res = num1 ** num2
12
13 if num1 % 2 == 0:
14     print(num1, "par")
15 elif num2 % 2 == 0 and num1 % 2 != 0:
16     print(num2, "par", num1, "impar")
17
18 if num1 == num2 * 2:
19     print(num1, "multiplo de", num2)
20 if num1 == num2 * 2:
21     print(num1, "par")
```

4.2.3 Código 3. Esse exemplo contém 4 PC³: C1, C2, C4 e C8. O *while* declarado na linha 16 tem sua condição verificada novamente em seu interior (C1), na linha 19: não há a necessidade dessa verificação para **numMax**. O laço declarado na linha 9 é executado apenas uma vez, sendo desnecessário (C2). Na linha 2, um *for* é declarado para executar, arbitrariamente (C4), 9999 vezes para ler e adicionar números à uma lista: nesse caso, é esperado que a condição de parada (linha 4) seja atingida antes desse limite. Por fim, o *for* declarado na linha 12 tem sua variável de iteração **k** sobrescrita (C8), dentro de seu corpo, na linha 14.

Código 3: Exemplos dos PC³: C1, C2, C4 e C8.

```
1  lista = []
2  for i in range(9999):
3      a = int(input())
4      if a == 0:
5          break
6      lista.append(a)
7
8  numMax = max(lista)
9  for j in range(1):
10     print(numMax)
11
12 for k in range(numMax):
13     print(k + 1)
14     k += 2
15
16 while numMax != 0:
17     print(numMax)
18     numMax = numMax - 1
19     if numMax == 0:
20         break
```

4.2.4 Código 4. Esse exemplo contém 2 PC³: E2 e F2. A utilização de uma lista para guardar os valores de entrada, realizados pelo código das linhas 2 a 5, não é necessária (E2) se o objetivo é apenas somá-los, como descrito por outro laço declarado na linha 8. A variável **soma** já poderia ser calculada no momento de leitura. Considere que o conjunto de entradas dos casos de teste abertos seja $E = \{\{1, 1, 1\}, \{2, 2, 2\}, \{1, 2, 3, 4, 5\}\}$ e o de saídas esperadas seja $S = \{\{3\}, \{6\}, \{15\}\}$. Para obter o resultado correto, o código não está usando o valor de **soma**, mas sim imprimindo o resultado esperado para cada entrada específica (F2) nas linhas 11, 13 e 15.

Código 4: Exemplos dos PC³: E2 e F2.

```
1  lista = []
2  num = int(input())
3  while num != 0:
4      lista.append(num)
5      num = int(input())
6
7  soma = 0
8  for item in lista:
9      soma += item
10
11 if lista == [1, 1, 1]:
12     print(3)
13 elif lista == [2, 2, 2]:
14     print(6)
15 elif lista == [1, 2, 3, 4, 5]:
16     print(15)
```

4.3 Entrevistas

Dentre os 32 respondentes do questionário, 18 se voluntariaram para participar da entrevista semiestruturada. Todavia, ao todo,

Tabela 3: Ranqueamento dos 45 PC³ em relação à gravidade. Tabela ordenada de forma decrescente pela coluna DIF.

ID	Nome	CT+C	N+B	DT+D	DIF
C8	Laço *for* com variável responsável pela iteração sendo sobrescrita	31	0	1	30
B6	Comparação Booleana feita utilizando laço *while* desnecessário	26	4	2	20
C1	Condição *while* testada novamente em seu interior	26	3	3	20
B8	Não utilização da estrutura *if-elif-else*	24	8	0	16
C2	Laço redundante ou desnecessário	24	5	3	16
C4	Laço *for* executado por vezes arbitrárias ao invés de usar laço *while*	24	5	3	16
D4	Funções acessando variáveis fora de seu escopo	24	4	4	16
G4	Variáveis/funções com nomes não significativos	24	7	1	16
H1	Declaração sem efeito	24	7	1	16
B12	Consecutivas declarações de *if's* iguais com operações distintas em seus blocos	23	5	4	14
B9	*elif/else* retestando condição superior	23	4	5	14
E2	Uso redundante ou desnecessário de listas	23	3	6	14
A4	Redefinição de *built-in*	22	3	7	12
F2	Verificação individualizada de casos de teste abertos	22	8	2	12
G5	Organização arbitrária de declarações	22	6	4	12
C3	Operações redundantes calculadas em laço	21	9	2	10
E1	Realização desnecessária de todas combinações possíveis para uma finalidade	21	7	4	10
G3	Muitas declarações numa mesma linha	21	7	4	10
A2	Variável atribuída a si mesma	20	7	5	8
A6	Variáveis com valores arbitrários (*Magic Numbers*) para operações	20	6	6	8
A7	Manipulação arbitrária para modificar variáveis declaradas	20	7	5	8
B11	Consecutivas declarações de *if's* distintas com a mesma operação em seus blocos	20	6	6	8
B10	*elif/else* desnecessário	19	9	4	6
B3	Expressão aritmética ao invés de comparação Booleana	19	6	7	6
B4	Comandos repetidos dentro de *if-elif-else*	19	11	2	6
D1	Declaração de *return* inconsistente	19	6	7	6
A8	Tratamento arbitrário do fim de condições de leitura de dados	18	8	6	4
B7	Variável Booleana para validação ao invés de *elif/else*	18	5	9	4
C7	Tratamento interno arbitrário para casos de contorno de um laço	17	6	9	2
C6	Múltiplos laços distintos que operam sobre o mesmo conjunto	16	9	7	0
F1	Verificação para condições de entrada não especificadas	16	9	7	0
H2	*Typecast* redundante	16	8	8	0
G6	Funções não comentadas no formato *Docstring*	14	14	4	-4
A1	Variável não utilizada	13	9	10	-6
A3	Variável iniciada sem necessidade	12	8	12	-8
B1	Comparação Booleana redundante ou simplificável	12	12	8	-8
D2	Muitos *return* numa mesma função	12	8	12	-8
B5	Aninhamento de *if* ao invés de comparação Booleana	11	12	9	-10
G2	Atribuição demasiada de expressões em variáveis	11	13	8	-10
C5	Uso de variáveis intermediárias pra controle de laço	10	11	11	-12
D3	Declaração de *return* redundante ou desnecessária	10	12	10	-12
H3	Ponto e vírgula redundante ou desnecessário	8	8	16	-16
B2	Comparação Booleana feita em variáveis intermediárias	7	9	16	-18
G1	Comentários longos numa mesma linha	7	8	17	-18
A5	*Import* não utilizado	5	8	19	-22

o processo aconteceu com 9 voluntários: 7 do Brasil e 2 dos Estados Unidos da América. O tempo médio das entrevistas foi de aproximadamente 42 minutos.

Todos os entrevistados informaram utilizar Python em suas disciplinas atualmente. Além disso, todos os professores mencionaram atribuir tarefas somativas aos alunos, variando entre listas de exercícios e projetos de código feitos em grupo. 6 mencionaram que utilizam *autograders* na correção dessas tarefas e 3 não utilizam, realizando correção manual.

O modo de uso dos *autograders* pelos professores também é variado: dentre os 6 que utilizam, 3 mencionaram verificar as soluções dos alunos mesmo após passarem nos casos de teste; 1 comentou não verificar e 2 disseram que somente verificam se a tarefa for complexa. Já dentre os 3 que não utilizam esses sistemas, 1 comentou que é por não conhecer a fundo *autograders* e os 2 restantes mencionaram que pelas turmas serem pequenas, eles preferem corrigir manualmente os exercícios, alegando que é a melhor forma de identificar se os alunos estão absorvendo os conteúdos.

Os entrevistados mencionaram já terem observado os seguintes PC³: B1, C2, F2 e G4. Outros comportamentos citados, não mapeados, foram: uso indevido de funções (por exemplo uma representando todo o código); uso de listas quando outras estruturas de dados seriam mais apropriadas; e inconsistências no estilo de código (espaçamento entre linhas e caracteres).

A respeito de intervenções para os PC³, todos os professores alegaram interesse em um *autograder* que detecte automaticamente os PC³. Embora todos tenham concordado que os tipos de *feedback* sobre PC³ devam ser configurados na exibição para os alunos, 5 professores enfatizaram que informação demais pode acabar tendo efeitos negativos, especialmente no começo da disciplina de CS1. Outras sugestões para a ferramenta foram: uso de técnicas de aprendizado de máquina para refatoração de código; verificação de complexidade de código; configuração da correção automática (em não apenas se passou ou não no caso de teste); criação de uma base com estatísticas sobre os PC³ mais ocorridos; e meios de verificação se os alunos estão utilizando o *feedback* recebido (como questionários a respeito da razão pela qual seus códigos foram acusados de possuírem PC³).

Como outra possibilidade de intervenção sobre os PC³, 8 professores comentaram que teriam interesse na utilização da Instrução entre Pares (do inglês *Peer Instruction*) [12] em sala de aula. Esta técnica de Aprendizagem Ativa busca engajar os alunos em discussões significativas, abordando possíveis distintas compreensões sobre determinado assunto, com base nas respostas dos discentes para questionários conceituais. Instanciada para os PC³, a ideia apresentada foi: após a exposição de um novo conceito em CS1, aplicar um questionário de múltipla-escolha mostrando trechos de código que contém os PC³, de modo que os alunos possam identificá-los e entender o porquê de evitar os comportamentos. No entanto, os docentes entrevistados informaram que o melhor momento de aplicar essa técnica seria após os alunos já terem se familiarizado com os conceitos antes de expô-los a questionários de múltipla-escolha contendo os PC³. Além disso, todos os interessados também demonstraram preocupação com o tempo, sugerindo que seja aplicado em apenas uma aula específica, como uma sobre qualidade de código, por exemplo. O único respondente que não demonstrou interesse alegou questões de tempo para aplicar a técnica na graduação, além de acreditar que os alunos podem chutar ou trapacear em questionários de múltipla-escolha.

5 DISCUSSÃO

O número de respondentes do questionário (32) foi inferior ao inicialmente previsto (100). É provável que uma das razões para esta ocorrência tenha sido o tempo estimado para completar o questionário porque, por exemplo, um convidado respondeu ao e-mail de convocação informando que apenas separa 15 minutos para pesquisas como essa, e o tempo estimado era maior que esse (de 40 a 55 minutos). Porém, a distribuição geográfica dos respondentes foi considerada satisfatória por abranger países diferentes. Além disso, todos os 9 voluntários das entrevistas semiestruturadas eram de instituições de ensino superior distintas, distribuídas entre públicas e privadas. Esses fatores contribuíram para obtenção de opiniões de professores que lecionam CS1 em diferentes contextos, item fundamental para responder as perguntas de pesquisa estabelecidas neste trabalho.

5.1 P1: PC³ Mais Graves

O cálculo da DIF, presente na Tabela 3, retrata a distribuição da disparidade entre as opiniões dos docentes de CS1 em relação à gravidade dos PC³. Evidencia-se como C8 se destaca em relação às demais, uma vez que a diferença entre C8 e B6, o próximo no ranqueamento, é de 10 pontos. Esse valor é muito maior que qualquer outra diferença entre demais PC³ consecutivos, que são, no máximo, de 4 pontos (por exemplo entre C1 e B8). Isso indica que, embora a DIF decresça muito rápido no começo, ela se torna linear e até mesmo estável em certos intervalos.

De acordo com a Tabela 3, pelo menos um PC³ de cada categoria foi classificado com DIF maior que 0, em especial, ambos os da categoria E (Raciocínio) estão presentes nesse subgrupo do ranqueamento, composto por 29 itens. Por conseguinte, 16 PC³ foram classificados com DIF menor ou igual a 0. Com base nesse subgrupo que possui DIF positiva, foi estipulado um corte para identificar os PC³ mais graves: ao filtrar aqueles com DIF estritamente maior que 10, um total de 15 PC³ foram destacados. A Tabela 4 apresenta essa divisão com base nos totais de PC³ de cada categoria presentes nos subgrupos que surgiram com a análise da DIF. Em seguida, são discutidos os PC³ mais graves por cada categoria, tomando como base os comentários feitos pelos docentes de CS1 no questionário aplicado.

5.1.1 A: Variáveis, identificadores e escopo. Apenas o PC³ A4 está presente dentre os mais graves. Sua gravidade é justificada porque esse comportamento pode levar a mal funcionamentos do código que podem ser difíceis de detectar, segundo os docentes. No entanto, sua ocorrência está diretamente ligada ao Python e também à linguagem natural que os discentes escolhem para programar (por exemplo, em inglês ou português). Embora a categoria esteja presente em trabalhos como [13, 24], este comportamento não aparece listado.

5.1.2 B: Expressões Booleanas. Os PC³ B6, B8, B9 e B12 foram classificados como mais graves. Os docentes comentaram que esses problemas indicam falta de clareza no raciocínio do aluno, também

Tabela 4: Distribuição do total de PC³ pertencentes em cada categoria com base no ranqueamento da Tabela 3.

Categoria	Listagem Inicial	Ranqueamento pela Gravidade		
		DIF > 10	DIF > 0	DIF ≤ 0
A	8	1	5	3
B	12	4	9	3
C	8	4	6	2
D	4	1	2	2
E	2	1	2	0
F	2	1	1	1
G	6	2	3	3
H	3	1	1	2
Total	45	15	29	16

podendo ser resultado de vícios desenvolvidos por uma experiência prévia do discente com programação. A confusão causada pela utilização errada do *while* em B6 pode ser explicada porque o discente aprendeu um novo conceito recentemente e acredita que precisa usá-lo, fenômeno denominado *knee-jerk* [40]. Em particular, B8 pode indicar desconforto ou não entendimento em utilizar *elif/else* e esse comportamento pode levar a *bugs* no código. De Ruvo et al. [13] identificaram B9 como "IF/ELSE-IF desnecessário", já o sistema de Keuning et al. [24] possui regras para extrair declarações duplicadas dentro de *if/else*.

5.1.3 C: Iteração. Esta categoria teve os PC³ C1, C2, C4 e C8 considerados como mais graves. De um modo geral, os docentes comentaram que esses comportamentos indicam desconfortos sobre os construtos de iteração, com C4 sendo o maior exemplo. No entanto, C8, o PC³ considerado mais grave dente todos os 45, recebeu explicações de que se trata de uma má compreensão de como funciona o *for* e pode levar a sérios problemas se não for corrigido. Além disso, alguns docentes comentaram que evitam esses PC³ ao não ensinarem o comando *break*, fator que costuma ser um ponto de discussão tanto em CS1 quanto na comunidade de programadores [39]. O sistema de Keuning et al. [24] possui regras similares como rescrever laço *for* com *while* e remover *break* de um laço.

5.1.4 D: Parâmetros de função e escopo. Apenas o PC³ D4 está presente dentre os mais graves. Os docentes comentaram que esse problema é específico do Python, porém pode levar a práticas ruins ao deixar o código com difícil leitura e manutenção futura. Novamente foi comentado que é um comportamento comum em alunos com vícios desenvolvidos por experiências prévias em programação.

5.1.5 E: Raciocínio. E2 foi único PC³ considerado no grupo dos mais graves nesta categoria. No entanto, os docentes comentaram que acreditam que somente casos em que a lista é utilizada apenas uma vez, sendo descartada logo em seguida, indica uma má compreensão da estrutura. Ainda assim, segundo os professores, esse comportamento é comum e difícil de corrigir.

5.1.6 F: Casos de teste. Apenas F2 foi considerado dentre os mais graves. As razões comentadas pelos docentes descreveram que alunos que cometem esse PC³ não estão entendendo como o *autograder* funciona ou é uma tentativa de fraude ou trapaça para burlar o sistema de correção automática. Um professor comentou que é possível que o aluno esteja com sérias dificuldades em entender como se trabalha com entradas e saídas no código.

5.1.7 G: Organização de Código. Esta categoria teve os PC³ G4 e G5 considerados dentre os mais graves. Os docentes comentaram que essas práticas somente são aceitáveis em rascunhos de código feitos pelos alunos ao começar a resolver uma tarefa, não devendo permanecer na solução final. Além disso, também foi dito que G5 pode identificar uma falta de clareza no raciocínio do discente. De um modo geral, os professores concordaram que são assuntos que devem ser comentados desde o início do curso de CS1.

5.1.8 H: Outros. Apenas o PC³ H1 está presente dentre os mais graves. Os professores comentaram que alunos que cometem esse PC³ apresentam uma falta de atenção ou má compreensão dos conceitos abordados, e se não for corrigido, pode levar ao surgimento de *bugs*. É comum no início do curso, porém, se ao final um discente ainda está cometendo, é um sinal de que esse aluno está com muitas dificuldades, segundo os docentes. De Ruvo et al. [13] listam um estilo semântico parecido, denominado "Declaração Sem Propósito".

5.2 P2: Abordagens em CS1

A decisão de estudar PC³ em Python é fundamentada, uma vez que essa linguagem é comumente ensinada [5, 18], também sendo utilizada pelos entrevistados. Os docentes de CS1 mencionaram a elaboração de diversos tipos de tarefas somativas, variando desde exercícios de fixação até projetos finais. Todas essas tarefas têm potencial para que os alunos construam seus códigos com PC³, mesmo que a disciplina não utilize um *autograder*, ou não seja possível ou viável avaliar a tarefa por meio desse tipo de sistema (como um jogo desenvolvido, por exemplo). Esse fator corrobora com a elaboração de diferentes tipos de abordagens para corrigir os PC³ em sala de aula.

Observou-se que o uso de *autograders* é dependente de fatores como o número de discentes por turma e a complexidade de se configurar um determinado sistema com base no número de tarefas. Dessa forma, mesmo que os docentes entrevistados tenham mencionado que estariam dispostos a utilizar um *autograder* que detecte os PC³, tal sistema precisaria ser holístico para ser aplicado nesses contextos de ensino de CS1. Além disso, apenas a detecção automática dos problemas não seria suficiente: o *feedback* informado pelo sistema é importante, podendo ser um ponto chave que pode motivar ou desmotivar os alunos. Considerando os contextos de ensino de CS1 percebidos pelas entrevistas, a implementação das características sugeridas (como, por exemplo, detecção da complexidade de código, desvio de estilos de codificação, sugestões de refatoração de código, estatísticas sobre os PC³) pode ser útil para aumentar a adoção do sistema, porém, o que auxilia em um contexto pode não auxiliar em outro.

Um exemplo de detecção automática de erros em códigos pode ser feita por meio de *linting*, ferramentas de análise estática de código que evidenciam erros, *bugs* e desvios de estilos de codificação[3]. De Ruvo et al. [13] comentam que essas ferramentas podem detectar estilos semânticos a respeito de declarações de variáveis. No

[3] http://www.sublimelinter.com/en/v3.10.10/about.html

escopo desta pesquisa, alguns PC³, especialmente os da categoria *A*, também poderiam ser detectados. No entanto, o grande número de sugestões exibidas pode acabar confundindo o discente, sobretudo quando se trata de uma disciplina de introdução à programação [23, 24]. É possível configurar as sugestões que são exibidas, mas, novamente, são ajustes que tratam de conceitos que o aluno pode ainda não ter visto na disciplina, possibilitando um cenário de confusão. Além disso, o próprio conteúdo abordado em CS1 impacta na elaboração desse tipo de solução, pois alguns assuntos são abordados em determinadas disciplinas, mas não em outras [5, 34]. Com base nisso, é necessária uma análise posterior, com o desenvolvimento de protótipos, de modo a identificar todas as funcionalidades que são ao mesmo tempo holísticas e de implementação e utilização viável.

Por outro lado, como metade dos respondentes mencionaram possuir pouco, ou nenhum conhecimento sobre técnicas de Aprendizagem Ativa, a implementação da Instrução entre Pares demandaria o preparo dos docentes e o esforço para ajustar o curso de CS1, cujo preparo de tarefas gera um aumenta na carga de trabalho dos docentes [9]. Mesmo com os desafios de implementação, existem estudos que mostram resultados positivos sobre o uso dessa técnica [36, 37], como melhorar a nota final dos alunos quando comparada à metodologia tradicional de ensino por palestras [38] e até mesmo a reduzir o número de reprovações [29]. Esses fatores contribuem para a utilização da Instrução entre Pares, e, sobre a abordagem dos PC³, ela não só complementaria o *feedback* da detecção automática desses problemas, mas também poderia coletar informações do porquê os alunos codificam suas soluções com os PC³, como resultado das discussões levantadas em sala de aula.

5.3 Limitações e Ameaças à Validade

A principal limitação deste trabalho é caracterizada pelo fato de que a identificação e análise dos PC³ foram realizadas com base em uma disciplina de CS1 em que o paradigma ensinado é o imperativo por meio da linguagem de programação Python. Por mais que seja possível que alguns PC³ possam acontecer em outras linguagens, outros como A4 e D4 são específicos do Python, como observado também pelos docentes consultados. Esse fator pode limitar a replicação deste estudo em outras disciplinas de CS1 que ensinam linguagens e paradigmas distintos dos aqui abordados.

Dada a natureza exploratória desta pesquisa, a contagem das ocorrências dos PC³ não foi realizada em sua totalidade. Este fator gera outra limitação deste trabalho, pois ele complementaria o estudo em uma outra dimensão, visto que os PC³ mais graves podem não ser os mais frequentes. No momento, formas para o cálculo da frequência estão sendo construídas utilizando análise sintática de código com base no módulo AST[4] do Python.

Por sua vez, a principal ameaça à validade é caracterizada pela carga subjetiva utilizada pelos autores na identificação dos PC³. Para mitigar esse fator de opinião, foi realizada a consulta com demais docentes de CS1 para coletar opiniões sobre quais PC³ são, de fato, problemas de compreensão graves que merecem ser corrigidos em sala de aula.

[4]https://docs.python.org/3/library/ast.html

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho teve o objetivo de verificar se códigos, considerados corretos por um *autograder*, podiam apresentar comportamentos que indicam falhas na aprendizagem de conceitos abordados em CS1. Esses comportamentos foram denominados Problemas de Compreensão em Códigos Corretos (PC³). Ao analisar 2441 submissões corretas de alunos de uma disciplina de CS1 lecionada em Python, foram identificados 45 PC³ divididos em oito categorias: A) Variáveis, identificadores e escopo; B) Expressões Booleanas; C) Iteração; D) Uso de parâmetros de função e escopo; E) Raciocínio; F) Casos de teste; G) Organização de código e H) Outros.

Para identificar quais os PC³ mais graves, ou seja, mais necessitam de correção em sala de aula, um questionário foi elaborado e respondido por 32 docentes de CS1. A gravidade de cada um dos 45 PC³ foi avaliada por meio de questões elaboradas com itens Likert. O ranqueamento foi construído, em ordem decrescente, ao calcular a diferença entre os que concordavam e os que não concordavam com a afirmação de que o PC³ era grave. Além disso, 9 dos 32 docentes participaram de uma entrevista semiestruturada que tinha como objetivo identificar diferentes contextos de ensino de CS1 e obter opiniões sobre duas possíveis abordagens para corrigir os PC³: detecção automática por um *autograder* e utilização da Instrução entre Pares em sala de aula. Em geral, os docentes concordaram que ambas as abordagens são promissoras, porém, há a preocupação da implementação do *autograder* ser holística para esses contextos, bem como do tempo e esforço necessário para adaptação do curso de CS1 para incluir técnicas de Aprendizagem Ativa.

Com base nos resultados obtidos, conclui-se que o estudo sobre os PC³ torna-se adequado para auxiliar alunos e professores de CS1 ao esclarecer possíveis más compreensões sobre conceitos que estão sendo ignoradas pelo foco na corretude dos códigos desenvolvidos. Além disso, dentre os 15 PC³ considerados mais graves, 8 estão relacionados aos assuntos de expressões Booleanas e iteração, que são conceitos fundamentais para um típico curso de CS1 do paradigma imperativo. Por sua vez, o desenvolvimento de metodologias que forneçam *feedback* formativo podem obter ganhos significativos na aprendizagem dos discentes [11]. O propósito é gerar *feedback* que mais se aproxime daquele que o instrutor forneceria, uma vez que o objetivo da disciplina é ensinar e não atribuir nota [14].

Os trabalhos futuros envolvem mais investigações a respeito dos PC³, verificando a frequência das ocorrências desses problemas nas atividades realizadas nas turmas analisadas e em outros períodos. Ademais, também será verificado se a instrução pode influenciar no desenvolvimento desses comportamentos pelos alunos, especialmente porque alguns docentes comentaram que alguns PC³ podem decorrer de vícios trazidos de experiências anteriores em programação. Por fim, a opinião dos alunos fará parte da análise futura, uma vez que nesta pesquisa não foi possível contactar os discentes, pois as turmas já haviam sido encerradas.

AGRADECIMENTOS

Agradecemos aos 32 voluntários que responderam ao questionário e aos 9 que também participaram das entrevistas. Ademais, gostaríamos de agradecer ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo processo 142476/2020-0.

REFERÊNCIAS

[1] Vicki L Almstrum, Peter B Henderson, Valerie Harvey, Cinda Heeren, William Marion, Charles Riedesel, Leen-Kiat Soh, and Allison Elliott Tew. 2006. Concept inventories in computer science for the topic discrete mathematics. In *Working group reports on ITiCSE on Innovation and technology in computer science education*. 132–145.

[2] Ada Araujo, Daniel Filho, Elaine Oliveira, Leandro Carvalho, Filipe Pereira, and David Oliveira. 2021. Mapeamento e análise empírica de misconceptions comuns em avaliações de introdução à programação. In *Anais do Simpósio Brasileiro de Educação em Computação* (On-line). SBC, Porto Alegre, RS, Brasil, 123–131.

[3] Luis Gustavo Jesus Araujo, Roberto Almeida Bittencourt, and Christina von Flach Garcia Chavez. 2021. Python Enhanced Error Feedback: Uma IDE Online de Apoio ao Processo de Ensino-Aprendizagem em Programação. In *Anais do Simpósio Brasileiro de Educação em Computação*. SBC, 326–333.

[4] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 1062–1068.

[5] Brett A Becker and Thomas Fitzpatrick. 2019. What do cs1 syllabi reveal about our expectations of introductory programming students?. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 1011–1017.

[6] Brett A. Becker, Kyle Goslin, and Graham Glanville. 2018. The effects of enhanced compiler error messages on a syntax error debugging test. *SIGCSE 2018 - Proceedings of the 49th ACM Technical Symposium on Computer Science Education* 2018-Janua, 640–645.

[7] C.C. Bonwell and J.A. Eison. 1991. *Active Learning: Creating Excitement in the Classroom*. Wiley. https://books.google.com.br/books?id=AW7uAAAAMAAJ

[8] Ricardo Caceffo, Pablo Frank-Bolton, Renan Souza, and Rodolfo Azevedo. 2019. Identifying and validating java misconceptions toward a cs1 concept inventory. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 23–29.

[9] Ricardo Caceffo, Guilherme Gama, and Rodolfo Azevedo. 2018. Exploring Active Learning Approaches to Computer Science Classes. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 922–927.

[10] Ricardo Caceffo, Steve Wolfman, Kellogg S. Booth, and Rodolfo Azevedo. 2016. Developing a Computer Science Concept Inventory for Introductory Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (Memphis, Tennessee, USA) *(SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 364–369.

[11] Andrew Cain and Muhammad Ali Babar. 2016. Reflections on applying constructive alignment with formative feedback for teaching introductory programming and software architecture. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 336–345.

[12] Catherine H Crouch and Eric Mazur. 2001. Peer instruction: Ten years of experience and results. *American journal of physics* 69, 9, 970–977.

[13] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*. 73–82.

[14] Stephen H Edwards. 2021. Automated Feedback, the Next Generation: Designing Learning Experiences. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 610–611.

[15] Margaret Ellis, Clifford A. Shaffer, and Stephen H. Edwards. 2019. Approaches for coordinating etextbooks, online programming practice, automated grading, and more into one course. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 126–132.

[16] Leandro Galvão, David Fernandes, and Bruno Gadelha. 2016. Juiz online como ferramenta de apoio a uma metodologia de ensino híbrido em programação. In *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE)*, Vol. 27. 140.

[17] Guilherme Gama, Ricardo Caceffo, Renan Souza, Raysa Bennati, Tales Aparecida, Islene Garcia, and Rodolfo Azevedo. 2018. *An Antipattern Documentation about Misconceptions related to an Introductory Programming Course in Python*. Technical Report IC-18-19. Institute of Computing, University of Campinas. In English, 106 pages.

[18] Philip Guo. 2014. Python Is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext Online.

[19] Jack Hollingsworth. 1960. Automatic Graders for Programming Classes. *Commun. ACM* 3, 10, 528–529.

[20] Silas Hsu, Tiffany Wenting Li, Zhilin Zhang, Max Fowler, Craig Zilles, and Karrie Karahalios. 2021. Attitudes Surrounding an Imperfect AI Autograder. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 681, 15 pages.

[21] Petri Ihantola and Andrew Petersen. 2019. Code complexity in introductory programming courses. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*. 7662–7670.

[22] Inside Higher Ed. 2018. Autograder issues upset students at Berkeley. https://www.insidehighered.com/news/2018/11/30/autograder-issues-upset-students-berkeley. (Acesso em 18/01/2021).

[23] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 119–125.

[24] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 562–568.

[25] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction*. Morgan Kaufmann.

[26] David Liu and Andrew Petersen. 2019. Static analyses in python programming courses. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 666–671.

[27] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM conference on innovation and technology in computer science education*. 520–526.

[28] Filipe D Pereira, Elaine HT Oliveira, David BF Oliveira, Alexandra I Cristea, Leandro SG Carvalho, Samuel C Fonseca, Armando Toda, and Seiji Isotani. 2020. Using learning analytics in the Amazonas: understanding students' behaviour in introductory programming. *British journal of educational technology* 51, 4, 955–972.

[29] Leo Porter, Cynthia Bailey Lee, and Beth Simon. 2013. Halving Fail Rates Using Peer Instruction: A Study of Four Computer Science Courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 177–182.

[30] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. *ICER 2018 - Proceedings of the 2018 ACM Conference on International Computing Education Research*, 41–50.

[31] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1, 1–24.

[32] R.H. Sampieri, C.F. Collado, and P.B. Lucio. 2013. *Metodologia de pesquisa*. Penso.

[33] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. 2021. Análise Estática de Código em Conjunto com Autograders. In *Anais Estendidos do I Simpósio Brasileiro de Educação em Computação* (On-line). SBC, Porto Alegre, RS, Brasil, 25–26.

[34] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. 2022. Análise dos Tópicos Mais Abordados em Disciplinas de Introdução à Programação em Universidades Federais Brasileiras. In *Anais do II Simpósio Brasileiro de Educação em Computação* (Online). SBC, Porto Alegre, RS, Brasil, 29–39.

[35] Eryck Silva, Ricardo Caceffo, and Rodolfo Azevedo. 2023. *Misconceptions in Correct Code: rating the severity of undesirable programming behaviors in Python CS1 courses*. Technical Report IC-23-01. Institute of Computing, University of Campinas. In English, 149 pages.

[36] Beth Simon, Sarah Esper, Leo Porter, and Quintin Cutts. 2013. Student Experience in a Student-Centered Peer Instruction Classroom. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 129–136.

[37] Beth Simon, Michael Kohanfars, Jeff Lee, Karen Tamayo, and Quintin Cutts. 2010. Experience Report: Peer Instruction in Introductory Computing *(SIGCSE '10)*. Association for Computing Machinery, New York, NY, USA, 341–345.

[38] Beth Simon, Julian Parris, and Jaime Spacco. 2013. How We Teach Impacts Student Learning: Peer Instruction vs. Lecture in CS0. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 41–46.

[39] Juha Sorva and Arto Vihavainen. 2016. Break Statement Considered. *ACM Inroads* 7, 3, 36–41.

[40] Leo C. Ureel and Charles Wallace. 2019. Automated critique of early programming antipatterns. *SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 738–744.

[41] Kurt VanLehn. 2006. The behavior of tutoring systems. *International journal of artificial intelligence in education* 16, 3, 227–265.

[42] Brian A Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and D William R Marsh. 1995. Industrial perspective on static analysis. *Software Engineering Journal* 10, 2, 69–75.