Avaliando a Construção do Conhecimento em Programação Através da Taxonomia SOLO

André Almeida[1], Eliane Cristina de Araújo[1], Jorge César Abrantes de Figueiredo[1]

[1]Universidade Federal de Campina Grande, Campina Grande, PB - Brasil

andrealmeida@copin.ufcg.edu.br, {eliane,abrantes}@computacao.ufcg.edu.br

Abstract. *A great challenge in the teaching of programming is evaluating the process of building this knowledge. Automated assessment tools evaluate the final produced code. However, the development process that involves choosing adequate structures to implement the problem solution can reveal a clear picture of the beginner's knowledge of programming. In this work, we seek to evaluate the students' progress considering the code complexity of their programming exercises using a framework that extends the SOLO taxonomy through the use of predefined programming building blocks. In the empirical study we have carried out, we evaluated the correlation between the mastery of basic design skills and the automated program assessment.*

Resumo. *Um dos grandes desafios no ensino de programação é avaliar o processo de construção deste conhecimento. Mecanismos de avaliação automática de programas avaliam o produto final gerado. Entretanto, o processo de construção envolve escolha de estruturas adequadas para implementar a solução do problema revela mais sobre o conhecimento do iniciante em programação. Neste trabalho, avaliamos o progresso dos alunos considerando a complexidade das respostas de seus exercícios utilizando um framework que estende a taxonomia SOLO através de blocos de construção dos programas. No estudo empírico realizado, exploramos a correlação entre o domínio das habilidades básicas de design e a avaliação automática dos programas.*

1. Introdução

No ensino de programação para iniciantes, sabemos que o desempenho dos alunos está fortemente correlacionado com a quantidade de programas que eles produzem [Araújo et al. 2013]. A avaliação automática (AA) de programas é um recurso didático que procura diminuir a carga sobre o professor, manter a consistência das correções e garantir que será dado rapidamente um *feedback* ao aluno. Entretanto há muitos aspectos a serem considerados na resolução dos problemas de computação que vão além da correção funcional, usualmente fornecida pelos avaliadores automáticos e juízes online. Estas questões vão desde a compreensão do que é o problema, definição de um plano de solução, implementação deste plano utilizando as adequadas construções de uma linguagem de programação e, por fim, a reavaliação da solução e realização de testes [Polya and Conway 2004]. De modo ortogonal a tudo isso, é necessário garantir a qualidade do software criado ao longo deste processo.

Os mecanismos de avaliação automática (AA) de programas costumam enfocar o produto final, ou seja, o código gerado pelo estudante. É importante observar que o processo de construção do programa pode revelar muito sobre o conhecimento do iniciante

DOI: 10.5753/cbie.sbie.2020.1813

em programação. Ele envolve a definição de estratégia e escolha de estruturas adequadas para a implementação da solução do problema. Os conceitos de programação, estruturas de dados, padrões de resoluções de problemas são revelados na estrutura interna do programa, no seu *design*. Estes elementos são reflexo da construção de conhecimento e da transformação ocorrida no aluno a partir de sua exposição às situações de aprendizado promovidas pelo professor ao longo do curso.

No contexto de disciplinas introdutórias de programação, uma abordagem para entender como alunos escrevem código é analisar a performance destes em exames e exercícios [Lopez et al. 2008, Sheard et al. 2008]. Diversos trabalhos identificam dificuldades enfrentadas pelos alunos, com destaque em problemas de compreensão de variáveis e atribuições [Jimoyiannis 2011], rastreamento de pequenos pedaços de código [Lopez et al. 2008], assim como no entendimento de como os alunos planejam a resolução dos exercícios [Costantini et al. 2020].

Em [Izu et al. 2016] as habilidades de *design* de código são examinadas através de um *framework* que usa a taxonomia SOLO [Biggs et al. 2014] para avaliar a complexidade de exercícios de programação. De maneira objetiva, a taxonomia SOLO (*Structure of Observed Learning Outcomes*) descreve as respostas que um aluno pode oferecer para uma tarefa. É um meio de classificar os resultados da aprendizagem em termos de complexidade, permitindo avaliar o trabalho dos alunos em termos de qualidade, em vez de quantas linhas ou bits de código estão corretos. Os autores deste trabalho adaptaram a taxonomia SOLO incluindo blocos de construção e conduzem uma análise detalhada do desempenho dos alunos.

Neste trabalho, procuramos avaliar o progresso dos alunos com base na estratégia proposta na literatura por [Izu et al. 2016]. Objetivamos ir além da verificação funcional do código final, exploramos o potencial da taxonomia SOLO para avaliar as habilidades dos alunos iniciantes em programação, considerando a resolução de problemas. Observamos os aspectos de construção do programa, contrastando com a compreensão abstrata dos conceitos de programação mapeados pela taxonomia. Utilizamos o mecanismo de AA para verificar os programas dos alunos e contrastar com a classificação obtida através da utilização do *framework*. Para tanto, foi conduzido um estudo empírico, com o proposito de analisar o desempenho dos alunos ao final do curso (após 15 semanas de instrução). Verificamos três exercícios de escrita de código que combinam os conceitos de vetores unidimensionais, *loops* e funções. No estudo, observamos os seguintes pontos:

- A capacidade de projetar e escrever o código corretamente com o objetivo de manipular vetores acima do nível não-estrutural da taxonomia SOLO;
- A consistência entre os níveis SOLO identificados para os três exercícios;
- A correlação entre o nível SOLO das respostas e o desempenho geral no curso, medido pelo resultado fornecido pela ferramenta de testes automáticos e pela média na disciplina.

Para avaliar os resultados deste processo, verificamos a classificação SOLO das respostas e a correlacionamos com o desempenho dos alunos no curso e com as avaliação automática realizada com as ferramentas empregadas no curso. Além disso, mapeamos os erros mais comuns que ilustram os desafios que os alunos enfrentam ao tentar combinar construções de programação de maneiras não triviais. No estudo correlacional, não encontramos fortes evidências de que as notas obtidas pelos estudantes na disciplina

correlacionam-se com o nível SOLO atingido pelo estudante nas questões. Entretanto, qualitativamente, é possível observar que os estudantes que obtém sucesso na disciplina são aqueles que atingem os níveis mais altos da taxonomia em suas questões.

2. Fundamentação Teórica

Nos cursos introdutórios de programação, a presença de ferramentas para avaliação automática é cada vez mais prevalente. Os sistemas de avaliação automática (AA), usualmente, fornecem *feedback* baseado nos resultados na aplicação de testes propostos pelos instrutores, ignorando a forma como ele é implementado. Melhorar o tipo de informação dado sobre a avaliação dos programas pode fazer com que os alunos iniciantes ganhem mais confiança e motivação para fazerem mais e melhores programas. Com esta motivação, buscamos explorar o potencial da taxonomia SOLO para avaliar as habilidades de *design* expressas nos códigos dos alunos e fazer um contraste com os resultados das avaliações obtidas através dos mecanismos de avaliação automáticas baseados em testes funcionais.

2.1. Estratégias para avaliação automática de programas

As ferramentas de avaliação automática (AA) têm ganhado papel de destaque nos desenhos didáticos dos cursos de programação introdutória. Elas viabilizam a produção automatizada de *feedbacks* frequentes, rápidos, padronizados e a um baixo custo. Elas permitem que os alunos possam fazer muitos exercícios, necessários para o domínio da programação [Araújo et al. 2013], e recebam resposta sobre a sua produção. Nos últimos anos, muita pesquisa foi feita nesta área [Ihantola et al. 2010] [Ala-Mutka 2005] [Douce et al. 2005]. Entretanto, a maioria dos AA, ao prover *feedback* aos alunos, consideram apenas os aspectos de correção funcional em detrimento da qualidade do código. Alguns trabalhos levam em consideração métricas de software como um aspecto relevante a ser avaliado nos exercícios de programação de alunos iniciantes. Ele foi referenciado pela primeira vez no trabalho de Mengel e Ulans [Mengel and Yerramilli 1999] e depois por Cardell-Oliver [Cardell-Oliver 2011]. As métricas de software são medidas extraídas do código do software que remetem a um critério de qualidade específico. Amplamente presente nas análises e processos da indústria de software, elas objetivam auxiliar a melhorar o processo de produção e o produto: o software [fen]. Mengel e Ulans propuseram que as métricas de software podem ser usadas como indicadores do desempenho dos alunos no aprendizado de programação. O trabalho de Singh, Gulwani e Solar-Lezama [Singh et al. 2013] propõe a detecção de erros e sugestão de como corrigí-los baseando-se na síntese de programas. Singh, Gulwani e Solar-Lezama consideram que os erros cometidos por alunos em suas submissões são previsíveis. Assim, propõe que o professor, além da especificação da questão (enunciado), dos testes e da solução de referência, forneça ainda um mapa dos possíveis erros que os alunos podem cometer. Este erros seriam modelados em uma linguagem definida para este propósito. A ferramenta de AA seria capaz de avaliar o erro cometido pelo aluno e propor uma possível correção com base no modelo de erros.

2.2. Avaliação com taxonomia SOLO

A taxonomia SOLO (*Structure of Observed Learning Outcomes*) é utilizada para avaliar o código do aluno e contrastar com os conceitos requeridos no aprendizado da programação.

É um meio de classificar os resultados da aprendizagem em termos de complexidade, permitindo avaliar o trabalho dos alunos em termos de qualidade [Izu et al. 2016]. Por ser uma taxonomia de objetivo geral, tornou-se popular no estudo de como os novatos manifestam sua compreensão e codificam a partir dos problemas de programação [Sheard et al. 2008, Shuhidan et al. 2009, Whalley et al. 2011]. Os níveis da taxonomia são definidos em função da compreensão e do esboço desta na resolução: primeiramente, os alunos podem escolher apenas um ou poucos aspectos da tarefa (nível não-estrutural), depois vários aspectos, mas que não estão relacionados (nível multi estrutural); depois, aprendem como integrá-los a um todo (nível relacional) e, finalmente, serão capazes de generalizar esse todo para aplicações ainda não ensinadas (resumo estendido), conforme a Tabela 1.

Tabela 1. Taxonomia SOLO revisada

Nível	**Projeto Algorítmico**
Pré-Estrutural (P)	Falta substancial de conhecimento dos componentes básicos (blocos de construção) e seu uso para resolver a tarefa especificada.
Não-Estrutural (U)	Uso de um bloco de construção ou modelo para resolver parcial ou completamente a tarefa especificada.
Multi Estrutural (M)	Modifica / Estende um bloco de construção ou combina sequencialmente dois ou mais blocos para resolver parcial ou completamente a tarefa especificada.
Relacional (R)	Combina e integra de maneira não simples dois ou mais blocos de construção para resolver a tarefa especificada.

3. Metodologia

No estudo desenvolvido neste trabalho, consideramos três exercícios aplicados na disciplina de Programação I do período letivo 2015.1. A realização de um estudo de caso retrospectivo, com dados históricos, foi necessária devido a impossibilidade de ter acesso aos alunos de forma presencial no corrente ano. Seguindo a proposta de [Izu et al. 2016], realizamos o levantamento dos blocos de construção básicos que consideramos que os alunos deveriam usar no nível não-estrutural após as 15 semanas de duração da disciplina. Os exercícios escolhidos exploram os conceitos de *loops* aninhados e vetores multidimensionais, mas por se tratar de uma disciplina introdutória, não consideramos que os alunos dominam com tranquilidade tais conceitos. Portanto, os blocos de construção referentes a estes conteúdos não são apontados na Tabela 2. Destacamos as operações em vetores, iteração por meio de *loops* simples e o uso de funções no contexto de Python.

Para coletar os dados, especialmente as respostas dos exercícios de programação, utilizamos a última submissão enviada pelos alunos à ferramenta de testes automáticos utilizada na disciplina. Foram selecionadas três questões aplicadas pelos professores e coletamos respostas de 38 alunos (o conjunto total composto pelos alunos que responderam pelo menos uma das questões), posto que a resolução destas era opcional, ou seja, os alunos poderiam responder ou não. Os exercícios selecionados para o estudo foram: JuncaoOrdenada, Rotaciona e LetrasCoincidentes. No primeiro, pede-se que a função receba duas listas e retorne uma lista com todos os elementos ordenados de forma crescente. Em Rotaciona, a função deve receber uma sequência e pede-se para fazer o deslocamento, de modo que os primeiros n elementos passam a ocupar as n posições finais da sequência.

Tabela 2. Lista de blocos de construção usados para a análise

Tipo	Nº	Descrição
Atribuição	1	Atribuição simples, por exemplo, x = 1
	2	Atribuição de um valor de uma expressão x = 4 * (a + 3)
	3	Sequência de atribuições x = 10; y = x + 1
Troca	4	Trocar o valor de duas variáveis a e b: a, b = b, a
Condicionais	5	Declaração If com condição simples
	6	Declaração if-else e elif
Condições Complexas	7	Duas ou mais condições combinadas por operadores lógicos (e, ou)
E/S Básicos	8	Aceitar entrada do usuário
	9	Apresentar saída por meio da função print
Loop for	10	*Loop* de 1 a n
	11	*Loop* com incremento/decremento
Loop Condicional	12	While(true) com declaração break
	13	While com uma condição simples (por exemplo, n>0) em que a variável testada foi inicializada e será atualizada no corpo
Acesso ao Vetor	14	Acesso pela localização A[3] = 4
	15	Adicionar elemento ao vetor V[x] = y
Operações com Vetores	16	Ler/Imprimir todos os elementos do vetor
	17	Encontrar min/max/soma de um vetor
	18	Atualização dos elementos
	19	Comparação entre elementos de um ou mais vetores
Funções	20	Utilização de cabeçalho da função e parâmetros def(parametro1, parametro2)
	21	Retorno da função: return
	22	Exibir o retorno da função print(funcao())

A última questão, Letras Coincidentes, consiste em encontrar as letras que coincidem na mesma posição em duas palavras dadas como entrada.

O conhecimento necessário para o correto desenvolvimento da solução dessas questões vai além dos exemplos simples de iteração (como adicionar os elementos de um vetor ou imprimir seus valores). Estas questões requerem o desenvolvimento de uma solução decompondo o problema em uma combinação de iteração, condicionais e comandos de atribuição; reutilizando e estendendo as construções e modelos de programação que aprenderam e praticaram. Portanto, os alunos deveriam trabalhar acima do nível não-estrutural, segundo a taxonomia SOLO.

Para aplicar o *framework* de avaliação da taxonomia SOLO revisada, identificamos os blocos de construção que os alunos usarão em seu *design*. Da mesma forma que em [Whalley et al. 2011], realizamos um processo de codificação iterativo para cada uma das questões para identificar os elementos mais importantes de cada tarefa. Por exemplo, na Tabela 3 listamos os blocos de construção que os alunos precisam usar e integrar para produzir uma resposta correta para a questão **JuncaoOrdenada**. De maneira similar, executamos o processo para as outras duas questões.

4. Resultados e Discussão

A distribuição das respostas dos alunos de acordo com a classificação SOLO pode ser observada na Tabela 4. É importante destacar que os códigos classificados como nível M (Multi Estrutural), não indicam necessariamente uma posição de inferioridade em relação aos classificados como nível R (Relacional). Isto pode acontecer, caso o aluno consiga

Tabela 3. Construções de programa necessárias para JuncaoOrdenada

Passo	Descrição
1	Iteração sobre as duas listas
2A	Construir a nova lista removendo e alternando(se for o caso) o menor elemento de cada lista
ou	
2B	Construir a nova lista apenas comparando e copiando o menor elemento da vez entre cada lista
ou	
2C	Construir a nova lista unindo os elementos das duas listas de entrada e então comparar para par, enviando para a esquerda os menores elementos

Tabela 4. Resumo das respostas SOLO

Nível SOLO	Juncao Ordenada	Rotaciona	Letras Coincidentes
R	4	0	0
M	14	21	9
U	1	3	4
P	0	0	0
Em branco	19	14	25

identificar uma maneira mais simples de combinar os blocos básicos. Isso aponta para a necessidade de selecionar cuidadosamente questões no contexto da exposição e experiência do aluno no curso; além disso, alguns alunos podem estar à frente se tiveram contato com alguma prática adicional de programação fora do curso.

Podemos observar que as respostas das turmas estão mais concentradas no nível multi estrutural, com algumas poucas estando nos níveis superior e imediatamente inferior. Para a questão JuncaoOrdenada, 10,53% das respostas foram classificadas no nível relacional, 36,84% no nível multi estrutural e 2,63% no nível não-estrutural. Para a questão Rotaciona, 55,26% das respostas foram classificadas no nível multi estrutural e 7,89% no nível não-estrutural. Por fim, para a questão LetrasCoincidentes, 23,68% das respostas foram classificadas no nível multi estrutural e 10,53% no nível não-estrutural.

Definindo um número para cada nível SOLO, de 1 (pré-estrutural) a 4 (relacional) e desconsiderando as questões em branco, temos um valor mediano igual a 3 para as três questões. Quanto a média, considerando um intervalo de confiança (IC) de 95% temos: 3.16 para JuncaoOrdenada com IC [2.89; 3.32], 2.91 para Rotaciona com IC [2.67; 3.04] e 2.69 para LetrasCoincidentes com IC [2.31; 2.85].

4.1. Análise da consistência do nível SOLO para os exercícios

Verificando as matrizes de distribuição utilizadas para uma visão geral da correspondência entre os níveis das respostas, percebemos que existem poucas ocorrências de respostas muito díspares.

Considerando a matriz entre JuncaoOrdenada x Rotaciona, apenas 5,3% dos alunos responderam uma questão no nível relacional e a outra no nível não-estrutural, sendo essa a única ocorrência de questões avaliadas em níveis que não são adjacentes. As demais, quando não respondidas no mesmo nível (considerando a soma da diagonal princi-

pal de cada matriz de distribuição e desprezando as questões em branco: 18,42% para JuncaoOrdenada e Rotaciona; 7,89% para JuncaoOrdenada e LetrasCoincidentes; e 5,26% para Rotaciona e LetrasCoincidentes), foram avaliadas em níveis adjacentes, revelando assim consistência entre as respostas.

4.2. Análise da correlação entre os níveis SOLO e os testes automáticos

Neste ponto estamos interessados em verificar a correlação entre os níveis SOLO das respostas e os resultados dos testes automáticos providos pela ferramenta de avaliação automática utilizada na disciplina. Considerando a última resposta enviada por cada aluno, foram executados 9 testes para a questão JuncaoOrdenada, 6 testes para a questão Rotaciona e 6 testes para a questão LetrasCoincidentes. No conjunto de dados coletados, foi atribuída a cada questão do aluno a média dos testes (nº de testes aceitos/nº total de testes), de maneira a representar os resultados com valores variando de 0 a 1. Assim, nos preocupamos em examinar se a média dos testes varia de acordo com a classificação SOLO apontada para cada resposta.

A Figura 1 apresenta a distribuição da média dos testes para cada nível SOLO, categorizando por questão. Podemos inferir que respostas nos níveis multi estrutural (M) e relacional (R) apresentam uma média mais próxima de 1, ou seja, todos os testes foram aceitos; enquanto que respostas no nível não-estrutural (U) apresentam uma maior variação. Analisando pelo coeficiente de correlação R de Spearman, temos resultados significantes para JuncaoOrdenada (R = 0,97), Rotaciona (R = 0,95) e LetrasCoincidentes (R = 0,99).

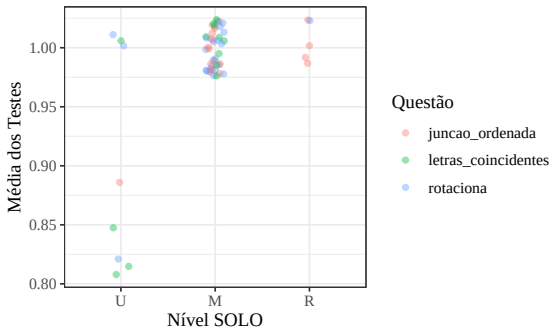

Figura 1. Correlação entre o nível SOLO e a média de testes aceitos

4.3. Análise da correlação entre as respostas SOLO e as médias na disciplina

Para esta última análise, consideramos a média alcançada pelos alunos na disciplina, composta por minitestes, prova, projeto e uma avaliação qualitativa. Dessa forma nos preocupamos em analisar a correlação entre os níveis SOLO das respostas dos alunos e o seu desempenho na disciplina. Como medida resumo, optou-se por utilizar a média SOLO. Para os alunos que responderam apenas um dos exercícios, a média SOLO foi dada pela avaliação dessa única resposta. Para os alunos que responderam dois ou os três exercícios, o resultado foi dado pela média aritmética da avaliação SOLO de cada resposta.

A Figura 2 denota a distribuição da média SOLO para a média da disciplina de cada aluno. Como observado, a correlação entre estas duas variáveis é muito fraca (R = 0,24) e não podemos dizer que, à medida que uma destas aumenta, a outra também

aumenta ou diminui. No entanto, podemos observar que a média SOLO para a maioria dos alunos está acima de 3 pontos: média geral igual a 2.98 com IC [2.85; 3.12] considerando 95% de confiança.

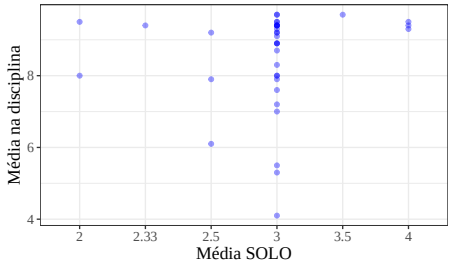

Figura 2. Relação entre a média SOLO e a média dos alunos na disciplina

4.4. Erros comuns cometidos pelos alunos

Em direção a uma avaliação qualitativa das habilidades de *design* de código dos alunos, identificamos os erros mais comuns cometidos pelos alunos.

Na questão JuncaoOrdenada, um erro comum no nível multi estrutural foi a redundância de código, através da inclusão trechos que em nada acrescentavam na resolução. Já no nível não-estrutural, o erro encontrado foi o uso de uma variável com valor muito grande para controlar o menor valor aceito nas listas de entrada. Assim, testando valores maiores que o definido, a função de unir os dois vetores ordenadamente não consegue inclui-los, resultando em erro. Devido a esse problema, avaliamos a questão nesse nível e consideramos que o código foi aceito para 8 dos 9 testes automáticos.

Na questão Rotaciona, o erro mais comum para esta questão foi a falta do *return* para as funções utilizadas, o que impedia que os retornos fossem visualizadas. Para estes casos, ainda consideramos que o código fosse aceito para todos os testes executados, porém avaliamos nesta categoria. O código é considerado como errado quando a saída do programa não é igual ao esperado, no entanto acreditamos que avaliar mais especificamente a lógica do programa para resolver o problema seja algo mais plausível. Uma outra resposta também avaliada no nível não-estrutural foi aceita para 5 dos 6 testes executados, uma vez que a operação de troca executada pelo aluno resultou em uma saída diferente da esperada devido a um erro de comparação.

Na questão LetrasCoincidentes, no nível não-estrutural, o erro mais comum foi o de *string index out of range*. Era necessário que o aluno identificasse que a comparação das letras das strings deveria ser feita em função da string de menor tamanho. Iterando entre qualquer uma das strings, caso esta fosse de maior tamanho, a comparação resultaria neste erro. Assim, apresentando esse erro, o código só foi aceito para 5 dos 6 testes executados.

4.5. Ameaças à Validade

Inicialmente, havíamos planejado a realização de um estudo controlado em laboratório, de maneira a observar as estratégias que os alunos escolhiam para resolver os exercícios e como evoluíam em direção a resposta. A definição das questões e seus blocos construtores, bem como os testes automáticos a serem realizados estariam em consonância.

Além disso, a definição da quantidade de alunos que fariam as questões poderia ter sido homogênea. No entanto, devido a impossibilidade de realização de experimentos presenciais, decidimos utilizar exercícios já aplicadas em laboratório devido a facilidade de coleta dos dados e disponibilidade no banco de dados de questões do ambiente de avaliação automática existente no curso. Estas questões podem ter impactado no resultado das notas de testes. Por este motivo, não podemos extrair conclusões pontuais e sim tendências.

A quantidade de questões em branco presentes na análise, superando os 50% do total de questões avaliadas uma vez que nem todos os alunos precisavam resolvê-las, pode impactar na expressabilidade dos resultados, visto que não se pode chegar a conclusões mais generalistas sobre o desempenho como um todo da turma. A coleta dos resultados para estas questões poderia ter um impacto bastante significativo.

5. Considerações Finais

Este trabalho se propôs a investigar a aplicação de um *framework* de avaliação das habilidades de *design*, assim como a completude e a corretude dos códigos de uma turma de alunos iniciantes, respondendo a três exercícios que requeriam manipulação de vetores, por meio da taxonomia SOLO. Os alunos apresentaram um desempenho contido entre níveis adjacentes da taxonomia, com poucas ocorrências de respostas em níveis muitos díspares: em média, para as três questões, 3,51% deles atingiram o nível relacional; 38,59% forneceram respostas no nível multi estrutural; e 7,02% no nível não-estrutural. A relação entre a média SOLO e a média na disciplina revelou que, para alguns casos, a variação nas médias da disciplina para uma média SOLO mais alta é menor. Enquanto que, com relação aos testes automáticos, a média dos casos aceitos foi acima de 83% para os três exercícios.

A análise deste estudo também nos ajudou a identificar os problemas que impedem os programadores iniciantes de usar e combinar blocos de construção para escrever trechos de código corretos. Poucos erros lógicos foram identificados nos níveis multi estrutural e não-estrutural. No entanto, pôde-se observar que descrição dos níveis SOLO em combinação com a lista de blocos de construção de um curso, de fato contribui para que os instrutores definam uma avaliação somativa e formativa que testa os alunos no nível certo de dificuldade, seja no contexto de exames ou em práticas em laboratório. Como nem todos os alunos progridem no mesmo ritmo, a lista deve refletir o nível mínimo necessário para acompanhar o curso.

6. Agradecimentos

Referências

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102.

Araújo, E., Gaudencio, M., Menezes, A., Ferreira, I., Ribeiro, I., Fagner, A., Ponciano, L., Morais, F., Guerrero, D., and Figueiredo, J. (2013). O papel do hábito de estudo no desempenho do aluno de programação.

Biggs, J., Collis, K., and Edward, A. (2014). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Elsevier Science.

Cardell-Oliver, R. (2011). How can software metrics help novice programmers? In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, page 55–62, AUS. Australian Computer Society, Inc.

Costantini, U., Lonati, V., and Morpurgo, A. (2020). How plans occur in novices' programs: A method to evaluate program-writing skills. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, page 852–858, New York, NY, USA. Association for Computing Machinery.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4–es.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, page 86–93, New York, NY, USA. Association for Computing Machinery.

Izu, C., Weerasinghe, A., and Pope, C. (2016). A study of code design skills in novice programmers using the solo taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, page 251–259, New York, NY, USA. Association for Computing Machinery.

Jimoyiannis, A. (2011). Using solo taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education*, pages 53–74.

Lopez, M., Whalley, J., Robbins, P., and Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. ICER '08, page 101–112, New York, NY, USA. Association for Computing Machinery.

Mengel, S. A. and Yerramilli, V. (1999). A case study of the static analysis of the quality of novice student programs. *SIGCSE Bull.*, 31(1):78–82.

Polya, G. and Conway, J. (2004). *How to Solve It: A New Aspect of Mathematical Method*. Penguin mathematics. Princeton University Press.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., and Whalley, J. L. (2008). Going solo to assess novice programmers. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '08, page 209–213, New York, NY, USA. Association for Computing Machinery.

Shuhidan, S., Hamilton, M., and D'Souza, D. (2009). A taxonomic study of novice programming summative assessment. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, page 147–156, AUS. Australian Computer Society, Inc.

Singh, R., Gulwani, S., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26.

Whalley, J., Clear, T., Robbins, P., and Thompson, E. (2011). Salient elements in novice solutions to code writing problems. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, page 37–46, AUS. Australian Computer Society, Inc.