Grouping and Differentiating Programming Exercises Using Minimal Programming Constructs

Ka Weng Pan*, Bryn Jeffries* and Irena Koprinska*

School of Computer Science, The University of Sydney, Sydney, NSW 2006, Australia

Abstract

Creating a sequence of learning activities for teaching programming to students is a challenging task. Educators must avoid overwhelming students with too many concepts, and need to cater to students who require a lot of repetition on each concept before progressing. We present a novel method to classify programming exercises based upon the set of minimal programming constructs, used by students in successful submissions. These constructs are automatically inferred from the abstract syntax tree representation of each code submission. Our method provides a meaningful measure of similarity between exercises, which can be applied to grouping exercises that require the same knowledge, recommending exercises requiring small increases in a student's existing knowledge, and identifying large gaps between consecutive exercises when creating a learning sequence for students. To demonstrate the utility of our method, we apply it to analysing data from two Python programming courses for high school students (beginners and intermediate), using the submissions of almost 10,000 students.

Keywords

Programming, K-12, abstract syntax tree, clustering

1. Introduction

Computer programming has been recognised as an important skill in today's technological society. Large-scale programming courses with thousands of students are increasingly being offered by universities and online platforms. These courses provide students with interactive course content and automatic feedback on programming exercises, improved accessibility and flexibility. However, students also experience difficulties for various reasons, e.g., instructional design, sequencing of learning activities, quality of teaching materials, learning environment and lack of sufficient feedback.

The data generated by such online platforms is rich but also large, complex, multi-dimensional and noisy, and therefore challenging to understand. Data mining methods can be used to automatically analyse this data and provide teachers with insights about student behaviour, in order to improve the course design, provide timely feedback to students, intervene at key points to prevent dropout or take other remedial actions.

In a programming course, students may become overwhelmed by the new syntax of a programming language. Once a student has mastered one language, concepts can often be transferred across to other languages [1]. Novice programmers, however, can struggle with the most basic of syntax elements [2]. One challenge in designing a course, then, is to choose the correct sequencing of material. Careful choice of the order in which concepts are introduced, minimising the cognitive load experienced by students, can significantly improve the success rate and student satisfaction of a course [3].

In this paper, we present a new data-driven method for grouping and differentiating programming exercises based upon a set of minimal programming constructs. These constructs are automatically inferred from the abstract syntax tree (AST) of the student submissions. We show how to extract these constructs, and based on them, define appropriate distance measures between exercises. We propose to use a map of primitive constructs to visualize the similarity between exercises, and clustering to group similar exercises. The utility of our method is demonstrated in the context of two large

CSEDM'25: 9th Educational Data Mining in Computer Science Education Workshop, July, 2025, Palermo, Italy

*Corresponding author.

kaweng.pan@sydney.edu.au (K. W. Pan); bryn.jeffries@sydney.edu.au (B. Jeffries); irena.koprinska@sydney.edu.au (I. Koprinska)

programming courses for high school students. Our method can be used to provide insights to teachers for understanding student behaviour, designing an appropriate sequence of exercises without large gaps in the required knowledge and recommending exercises with suitable increase in existing student knowledge, or for identifying key intervention points - e.g., exercises with overly complex solutions than intended, which could benefit from more guidance.

2. Background and Related Work

The use of syntax elements to identify specific concepts in assessment tasks has been explored previously. A working group [4] reviewed the concepts commonly covered in novice programming courses and linked them to the syntax elements that appeared in associated multiple-choice questions using Java or Python. They demonstrated how dependent concepts could be assessed in a more granular form, facilitating pedagogical approaches such as Mastery Learning. We extend this thinking by looking at student submissions to programming exercises, rather than the code included in the assessment task.

There is a large body of research on data-driven methods for analysing student log data, e.g. on clustering students to understand behaviour over time [5, 6], find the characteristics of high and low performing students [7, 8], analysing student difficulties [9], automatic hint generation [10, 11], early prediction of student progress, final course mark and dropout [12, 13, 14].

Clustering student code submissions has also been explored. Joyner et al. [15] used the commercial system Sense to cluster code submissions from an introductory programming MOOC, with the goal of providing feedback to teachers for course revision. The results are analysed in terms of percentage of students using specific constructs and methods, e.g., for-loops, find(). Glassman et al. [16] developed OverCode - a system that groups similar programming solutions and allows teachers to visualize and explore the similarities and differences of these solutions, and develop a high-level view of student understanding and misconceptions. Paiva et al. [17] proposed AsanasCluster for clustering correct student submissions based on semantic features extracted from the control flow and data flow graphs, which capture key aspects of the algorithmic strategy rather than syntax. Effenberger and Pelanek [18] focused on generating interpretable clusters of student submissions. They manually selected a small subset of features for each programming exercise, combined them as frequent patterns using a pattern matching algorithm and provided a summarization and visualization of each group. In contrast, our goal is to explore automatic extraction of programming constructs.

Our work is closely related to the sub-area of domain modelling which aims to map *problems* (exercises, questions) to *concepts* (skills, knowledge components) that they assess. This mapping is typically encoded in a Q-matrix [19] and used to assess which concepts are learned by the students in a given problem. Once the matrix is defined, methods based on knowledge tracing such as Bayesian knowledge tracing [20], performance factor analysis [21] or deep knowledge tracing [22] can be used to determine the student's mastery of the skills and provide recommendations to students.

Methods for automatically deriving Q-matrices from student log data have been proposed. For example, Barnes [23] utilised random matrix initialization and hill-climbing algorithm, Desmarais et al. [24] used an initial Q-matrix designed by experts which was refined with the alternating least squares algorithm, and Picones et al. [25] proposed an entirely data-driven method combining hill climbing to derive the initial Q-matrix and alternating least squares algorithm to refine it. However, data-driven methods lack interpretability since it is not clear what the concepts represent — e.g., which concept represents "if-statements" or "for-loops".

A neural network based approach for discovering knowledge components in an introductory Java course was proposed in [26]. The network was trained to predict the correctness of student programming submissions, encoded using the `code2vec` model. The learned hidden layer vectors were interpreted as knowledge components and analysed by experts. It was found that the discovered knowledge components were different compared to how experts were likely to define them — they were amalgamations of micro-concepts, either exercise-specific or applicable to multiple exercises.

Our task is different — our goal is not to predict the correctness of student programs but to group programming exercises based upon a minimum set of programming constructs, which can be seen as a set of knowledge components. The programming constructs are inferred automatically from the AST representation of student programs and their meaning is clearly defined and interpretable. Our overall goal is to help educators to improve the learning outcomes by designing an appropriate sequence of programming exercises, identifying the large gaps between consecutive exercises in terms of programming constructs students use, understanding which concepts students are struggling with and recommending appropriate exercises and actions.

3. Method

We propose a method for grouping programming exercises based on the set of minimal primitive programming constructs used to successfully complete these exercises. It consists of three steps:

1. Extracting primitive construct sets from successful submissions
2. Identifying the set of minimal primitive constructs for each exercise through aggregation
3. Defining the distance between exercises

3.1. Extracting Primitive Construct Sets

We first generate the corresponding AST for each correct submission using the `ast.parse` function in Python. The correct submissions are the ones which have passed all test cases. Each node class in the `ast` module provides information about the type of the programming construct. Hence, we initialise the set of primitive constructs for a submission to be the distinct set of node classes within its AST.

We then refine the set of primitive constructs through the following steps:

1. Extend the set with the name of the library functions called by masking self-defined function names within the AST and obtaining the remaining function labels.
2. Obtain further contextual information for the set of primitive constructs by including the exception types caught by any exception handlers used, the constant types used, and the variable operations performed.
3. Transform AST primitives to show any hidden hierarchical relationships or associations between primitives. For example, generalising `ast.Eq` to (`ast.Compare`, `ast.Eq`) indicates the association between the equality operator and comparisons.
4. Clean the set of extracted node classes by eliminating node classes that are wrappers of primitive constructs (e.g., `ast.Module` and `ast.Expr`), and classes that are redundant (i.e., have been recaptured during the extended extraction and transformation stage).
5. Relabel `ast` node classes to improve interpretability.

3.2. Identifying the Set of Minimal Primitive Constructs for Each Exercise

Let $S = \{s_1, \ldots, s_n\}$ be the set of correct submissions for an exercise p and X_i be the set of primitive constructs used by submission $s_i \in S$. We define the set of minimal primitive programming constructs for exercise p as

$$M_p = \bigcup_{i=1}^{n} X_i^* \quad \text{where } X_i^* = \begin{cases} X_i & \text{if } X_j \not\subset X_i \text{ for all } s_j \in S \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

That is, the set of minimal primitive constructs for each exercise is the union of the primitive construct sets of passing (successful) submissions that are not a proper superset of any other passing submission's set of primitive constructs. This aggregation method aims to capture a holistic representation of the multiple sets of programming constructs that could generate a successful solution.

	Beginners Course		Intermediate Course	
Rank	Primitive construct	Freq. (max 40)	Primitive construct	Freq. (max 25)
1	[variable] load	40	assignment	25
2	function call	40	[constant] str	25
3	[constant] str	36	[imported function call] print	25
4	[variable] store to	32	[variable] load	25
5	assignment	29	function call	25
6	[imported function call] print	28	[variable] store to	25
7	[constant] int	27	[imported function call] input	25
8	[imported function call] input	23	attribute	24
9	if/elif/else statement	16	[constant] int	24
10	function argument/arguments	16	if/elif/else statement	24

Table 1
Frequency of the most common constructs in the minimal primitive construct sets for the Beginners and Intermediate courses

Exercise	Minimal Primitive Construct Set
1	constant: (str), function call, imported function call: (print), variable: (load)
2	constant: (str), function call, imported function call: (print), variable: (load)
3	assignment, constant: (str), function call, imported function call: (print), variable: (load, store to)
4	assignment, constant: (str), function call, imported function call: (print), variable: (load, store to)
5	assignment, constant: (int), function call, imported function call: (print), variable: (load, store to)
6	assignment, binary operation: (add, sub), constant: (int), function call, imported function call: (print), variable: (load, store to)

Table 2
Minimal primitive construct sets for the first six exercises in the Beginners course

3.3. Defining Distance Between Exercises

Based on the set of minimal primitive constructs, we define two distance measures between exercises: 1) consecutive distance between two exercises and 2) cumulative distance across a sequence of exercises.

Distance between two exercises. Given two exercises p_x and p_y and their corresponding set of minimal primitive constructs M_x and M_y, we define the distance from exercise p_x to exercise p_y as

$$D_{p_x p_y} = M_y - M_x \tag{2}$$

i.e., the constructs that are seen in p_y but not in p_x. Hence, $D_{p_x p_y}$ is based on the estimated number of new minimal constructs required to complete p_y with the prior knowledge of the constructs used in p_x.

Cumulative distance across an exercise sequence. The distance between two exercises from (2) can be extended to define a cumulative distance across an exercise sequence.

Given an exercise sequence $P = \{p_1, \ldots, p_n\}$, and assuming that the students complete the exercises in this order, the distance from p_{k+1} to the prior sequence $\{p_1, \ldots, p_k\}$ is computed as

$$D_{p_{k+1} p_{1\ldots k}} = M_{k+1} - \bigcup_{i=1}^{k} M_i \tag{3}$$

i.e., this is the distance to progress to the next exercise p_{k+1} with the understanding of the set of minimal constructs used in all previous exercises.

These two distance measures are used in our analysis to identify insights into exercise sequencing and their impact on student performance.

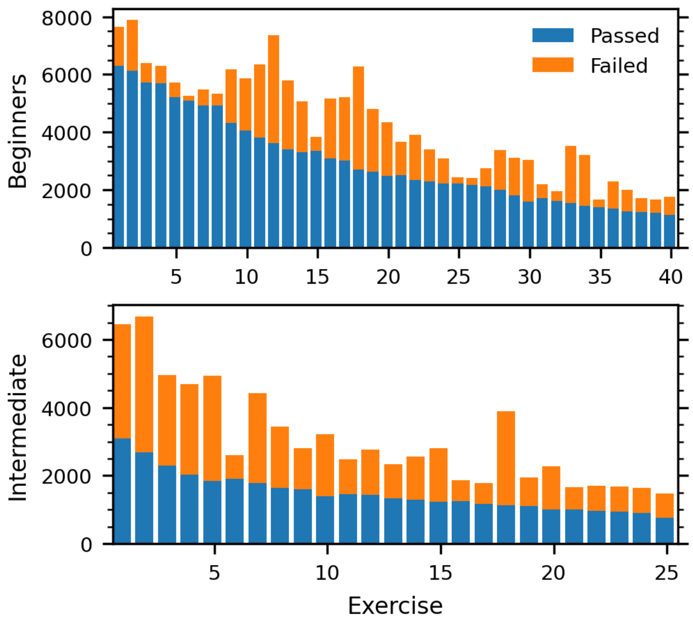


Figure 1: Number of passed and failed submissions for each exercise in the Beginners and Intermediate courses

4. Data

We applied our method to two online Python programming courses for high school students in Australia, that ran for five weeks. The online learning platform offered several courses for different levels of ability. We used data from two courses: Beginners and Intermediate. The Beginners course is for Year 7–8 students learning fundamental computational thinking, and the Intermediate course is for Year 9–10 students learning data structures, files and functions. A new set of exercises is released every week (8 for Beginners and 5 for Intermediate) and can be completed until the end of the course. Students submit their solution for automated marking and feedback based on test cases. They can correct their solution based on the feedback and resubmit multiple times until they pass all tests.

The Beginners course had 40 exercises and was attempted by 6,539 students, while the Intermediate had 25 exercises and was attempted by 3,288 students. Most students did not complete all exercises. For the attempted exercises, the mean number of submissions per student was 1.47 for Beginners and 2.00 for Intermediate. The number of passed and failed submissions for each exercise is shown in Figure 1.

The data was provided in anonymised format, with all student details removed and student identifiers substituted with randomly generated numbers to prevent re-identification.

5. Results and Discussion

5.1. Identified Constructs

Our method identified 102 base primitive constructs across the minimal construct sets generated from the passing submissions for the 40 exercises in the Beginner course, and 154 constructs for the 25 exercises of the Intermediate course. The ten most common primitive constructs for each course are listed in Table 1. It can be seen that the items in each set are the same except for `function arguments` which appears in Beginners only and `attribute` which appears in Intermediate only. Only the top two items from Beginners appear in all 40 exercises, while the top seven items for Intermediate appear in all 25 exercises for this course. This may reflect the need to routinely use a larger common set of concepts in the more advanced Intermediate course, where students need to combine concepts in different ways, while the Beginners course introduces concepts in a more granular manner.

Exercise	assignment	binary operation		constant		function call	imported function call	variable	
		add	sub	int	str		print	load	store to
1	0	0	0	0	1	1	1	1	0
2	0	0	0	0	1	1	1	1	0
3	1	0	0	0	1	1	1	1	1
4	1	0	0	0	1	1	1	1	1
5	1	0	0	1	0	1	1	1	1
6	1	1	1	1	0	1	1	1	1

Table 3
Primitive construct map for the first six exercises in the Beginners course

5.2. Per-Exercise Minimal Primitive Construct Sets

The set of minimal primitive constructs represents the constructs used by students to solve an exercise. It can serve as a summary of the primitives that a new student will need to complete the exercise.

The minimal primitive construct sets for the first six exercises of the Beginners course are shown in Table 2. In the first exercise, students simply need to modify the printed string of a single-line program, to the expected solution: `print('Hello, World!')`. The set therefore only includes the four syntax constructs relating to printing a string literal. The set for the subsequent exercises grows to include handling of string variable (in Exercise 3) and then integers (Exercises 5 and 6).

By vectorising each exercise based on the constructs present within their minimal primitive set, we can create a *primitive construct map* that shows the similarities between exercises. We explored the utility of this technique for the two courses. Table 3 shows the map generated for the first six exercises from the Beginners course as listed in Table 2. We can see that the first two exercises are syntactically identical, requiring the same primitive constructs. This was intentional in the activity design: Exercise 2 was similar to Exercise 1, requiring students to write a single-line printing program.

Exercises 3 and 4 were also intentionally paired and required similar solutions (Exercise 3 involved multiple `print` operations with the same string variable, while Exercise 4 used two string variables), and this is reflected in the primitive map. Similarly, the transition from Exercise 4 to Exercise 5 includes only one new primitive (integer constants), with all other primitives having been used multiple times before, making this a very gradual introduction.

The transition from Exercise 2 to 3 introduces both the use of a variable reference to store a value (`ast.Store`) and the assignment of that value (`ast.Assign`). While this was captured as two primitives, they are so closely related and naturally occur together. This transition therefore does not appear to require improvement. On the other hand, it can be seen from the map that Exercise 6 introduces both addition and subtraction for the first time, which may be especially challenging for students who do not have a strong mastery of arithmetic. The sequencing could be improved by introducing addition first, and postponing the need for subtraction to a later exercise.

The minimal primitive construct sets for the Intermediate course were substantially larger than for the Beginners, as can be seen from the results for the first four exercises, shown in Table 4. Each set of constructs was surprisingly large, considering what was expected by the exercise authors. For instance, Exercise 1 required the student to write a program that spelled out the letters of an input word as in a spelling bee competition. A reference solution that would have generated the required output was:

```
word = input('The next word is... ')
print(word.lower())
for letter in word.upper():
    print(letter)
print(word.lower())
```

This program would have used only the eleven items shown in bold for Exercise 1 in Table 4. The

Exercise	Minimal Primitive Construct Set
1	**assignment**, **attribute**, augmented assignment: (add), binary operation: (add), comparison: (lt, noteq), comprehension, constant: (int, **str**), function argument/arguments, **function call**, imported function call: (**input**, join, len, list, **lower**, map, **print**, **upper**), keyword argument, lambda, list comprehension, loop: (**for loop**, while loop), starred, subscript, variable: (**load**, **store to**)
2	assignment, attribute, augmented assignment: (add), binary operation: (add), boolean operation: (and), comparison: (eq, gt, in, noteq), constant: (bool, int, str), f-string, function argument/arguments, function call, if-then-else expression, if/elif/else statement, imported function call: (format, input, int, islower, isupper, lower, print, upper), keyword argument, lambda, tuple, unary operation: (not), variable: (load, store to)
3	assignment, binary operation: (add), comparison: (gt, gte, lt, lte), constant: (int, str), f-string, function argument/arguments, function call, functiondef, if-then-else expression, if/elif/else statement, imported function call: (float, input, int, print), return, variable: (load, store to)
4	assignment, attribute, binary operation: (add, div, mod, mult, sub), comparison: (in, lte), constant: (float, int, str), excepthandler: (ValueError), f-string, function argument/arguments, function call, functiondef, if/elif/else statement, imported function call: (exit, float, format, input, int, len, print, replace, round, str), keyword argument, return, try, variable: (load, store to)

Table 4
Minimal primitive construct sets for the first four exercises in the Intermediate course

Minimal Primitive Construct Set	Exercises
constant: (str), function call, imported function call: (print), variable: (load)	1, 2
assignment, constant: (str), function call, imported function call: (print), variable: (load, store to)	3, 4
assignment, comparison: (gt, gte, lt, lte), constant: (int, str), function call, if-then-else expression, if/elif/else statement, imported function call: (input, int, print), variable: (load, store to)	13, 14

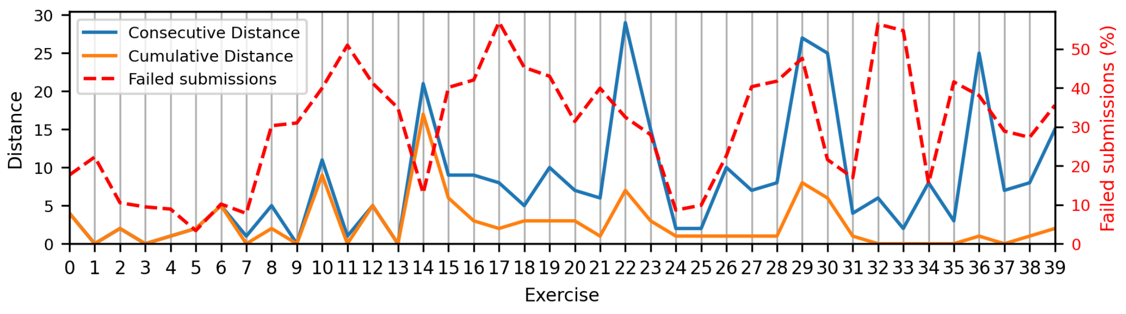
Table 5
Clusters of exercises sharing the same minimal primitive construct set for the Beginners course

other eighteen items that appear in the minimal primitive construct set demonstrate the diversity of approaches taken by students. Using very different syntax primitives than the expected requires attention - there may be a need to improve the course design and provide better guidance and resources.

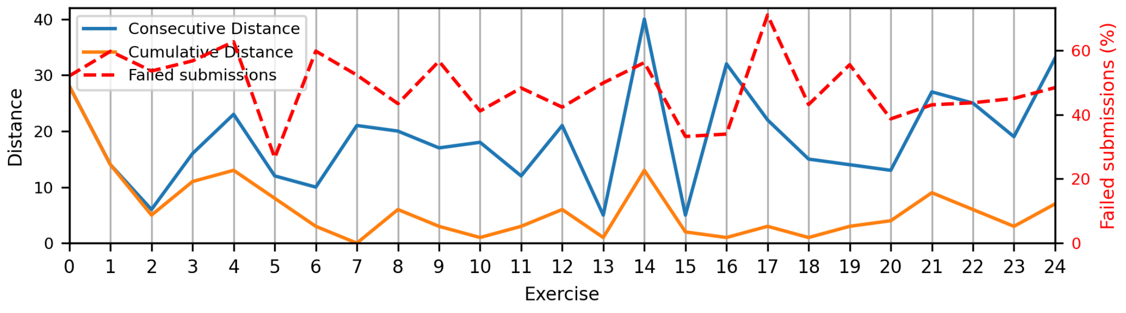
5.3. Clusters Based on the Set of Minimal Primitive Constructs

It is desirable to group existing exercises according to the primitive constructs required to solve them. The pool of similar exercises can be used to recommend exercises to students needing further practice. To explore the feasibility of this approach, we formed clusters where all exercises within a cluster shared the same set of minimal primitive constructs.

For the Beginners course, we identified 37 clusters. Of these, 34 were trivial, consisting of a single exercise, while the other three contained a pair of exercises, as shown in Table 5. Each of these pairs was intentional, with the first exercise intended to provide scaffolding for a approach that the student was expected to use unaided in the second exercise. For the Intermediate course, we identified 25 clusters, one for each exercise. Since the exercises in this course were each designed to cover different material, this was unsurprising. Hence, these results validated our expectations and showed the feasibility of this approach. However, the clustering can be improved. As we noted previously, the set of minimal primitive constructs can be significantly larger than the intended solution. In such cases, effective clustering may require pruning less frequent members of the minimal primitive constructs set, or employing some level of tolerance when sharing constructs.



(a) Beginners



(b) Intermediate

Figure 2: Exercise distance compared to failure rate of submissions for (a) Beginners and (b) Intermediate

5.4. Comparison Between Exercise Distance and Exercise Failure Rate

We hypothesised that an exercise that required a large number of new programming constructs would have poor outcomes, as students encounter significant cognitive load applying these new constructs. To explore this, we compared the consecutive and cumulative exercise distances $D_{p_i p_{i-1}}$ and $D_{p_i p_{1 \ldots i-1}}$ with each exercise's failure rate, as shown in Figure 2. The exercise failure rate is the proportion of all syntactically valid submissions which failed any of the correctness tests.

For the the Beginners course the failure rate correlated poorly with the cumulative distance (Spearman's coefficient $r_s = 0.06$) and weakly with the consecutive distance ($r_s = 0.31$). The absence of a strong correlation is not surprising, since we would expect many factors to influence the difficulty of an exercise. However, it is interesting to look at cases where there was a large jump in new constructs in an exercise and compare it with the failure rate. For instance, Exercises 14 and 22 involved a large number of new constructs and low failure rate. These exercises used several functions from the `turtle` graphics library. Despite their large number, these constructs were intuitive to use and explained clearly in the preceding course content, which may be the reason for the low failure rate.

On the other hand, Exercise 29 also shows a high jump in the number of constructs but also a high failure rate. Upon inspection, we found that many submissions involved finding the maximum-valued item in a list using functions like `max` or `sort` that had not been introduced yet. Additionally, we observed that many submissions used a wide variety of constructs that had been used in earlier exercises, such as loops and inequality operators. This suggests that students would benefit from more practice and explanations before attempting this exercise and also better scaffolding towards the intended solution which was far simpler.

In contrast, for the Intermediate course, the failure rate correlated more with the cumulative distance ($r_s = 0.26$) than the consecutive ($r_s = 0.07$). Even so, the consistently high failure rate for all exercises (mean 48.7%), together with the high mean consecutive distance of 18.7 constructs, suggests a large diversity of approaches. This exercise could also benefit from providing more practice and guidance.

6. Conclusion and Future Work

In this paper, we presented a novel method to group programming exercises based upon the programming constructs commonly used by students in successful submissions. It mines the abstract syntax tree of submissions, to determine the set of minimal primitive constructs required for each exercise. A map of primitive constructs was used to visualize the similarity between exercises and clustering was used to group similar exercises. We introduced two distance measures to quantify the number of new primitives required to complete the next exercise, having completed the previous exercise and all previous exercises, and explored their relation to the exercise failure rate.

The proposed method was applied to two large-scale online Python programming courses undertaken by almost 10,000 high school students. It was able to successfully identify flaws in the exercise design and key points where more practice and guidance would be beneficial, as well as groups of similar exercises that can be used for recommending exercises with a small increase of knowledge for practice and revision. It also identified exercises with overly complex solutions, including the use of new, not studied constructs, that would benefit from providing better scaffolding and intervention.

We found that the set of minimal primitive constructs was surprisingly large, especially for the Intermediate course, due to the diversity of approaches taken by students, perhaps due to using external resources. In future work we plan to refine our method to exclude outlier constructs and provide a tighter and more representative set. It is expected that this would improve the grouping of exercises and allow to recommend new exercises with few new constructs. We also intend to go beyond the primitive constructs and explore ways to automatically extract more complex constructs expected in advanced programming courses, such as nested loops and recursion.

Declaration on Generative AI

No Generative AI was used during the preparation of this paper.

References

[1] E. Tshukudu, Q. Cutts, Understanding conceptual transfer for students learning new programming languages, in: Proceedings of the ACM Conference on International Computing Education Research, 2020, pp. 227–237.

[2] B. Jeffries, J. A. Lee, I. Koprinska, 115 ways not to say hello, world! syntax errors observed in a large-scale online CS0 Python course, in: Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education (ITiCSE), ACM, 2022, pp. 337–343.

[3] R. Mason, C. Seton, G. Cooper, Applying cognitive load theory to the redesign of a conventional database systems course, Computer Science Education 26 (2016) 68–87.

[4] A. Luxton-Reilly, B. A. Becker, Y. Cao, R. McDermott, C. Mirolo, A. Mühling, A. Petersen, K. Sanders, Simon, J. Whalley, Developing assessments to determine mastery of programming fundamentals, in: Proceedings of the ITiCSE Conference on Working Group Reports, ACM, 2018, p. 47–69.

[5] J. McBroom, K. Yacef, I. Koprinska, DETECT: A hierarchical clustering algorithm for behavioural trends in temporal educational data, in: In Proceedings of the International Confernce on Artificial Intelligence in Education (AIED), Springer, 2020, pp. 374–385.

[6] R. Cerezo, M. Esteban, J. C. Sánchez-Santillán, Nunez, Procrastinating behavior in computer-based learning environments to predict performance: A case study in Moodle, Frontiers in Psychology 8 (2017).

[7] I. Koprinska, J. Stretton, K. Yacef, Predicting student performance from multiple data sources, in: Proceedings of the International Conference on Artificial Intelligence in Education (AIED), 2015, pp. 678–681.

[8] J. McBroom, B. Jeffries, I. Koprinska, K. Yacef, Mining behaviors of students in autograding

submission system logs, in: In Proceedings of the International Conference on Educational Data Mining (EDM), 2016.

[9] J. McBroom, B. Paassen, B. Jeffries, I. Koprinska, K. Yacef, Progress networks as a tool for analysing student programming difficulties, in: In proceedings of the Australasian Computing Education Conference (ACE), ACM, 2021, p. 158–167.

[10] J. McBroom, I. Koprinska, K. Yacef, A survey of automated programming hint generation: The HINTS framework, ACM Computing Surveys 54 (2021) 1–27.

[11] T. W. Price, Y. Dong, R. Zhi, B. Paassen, N. Lytle, V. Cateté, T. Barnes, A comparison of the quality of data-driven programming hint generation algorithms, International Journal of Artificial Intelligence in Education 29 (2019) 368–395.

[12] V. Zhang, B. Jeffries, I. Koprinska, Predicting progress in a large-scale online programming course, in: In proceedings of the International Conference on Artificial Intelligence in Education (AIED), Springer Nature, 2023, pp. 810–816.

[13] S. Polito, I. Koprinska, B. Jeffries, Exploring student engagement in an online programming course using machine learning methods, in: Proceedings of the International Conference on Artificial Intelligence in Education (AIED), Springer, 2022, pp. 546–550.

[14] V. Dsilva, J. Schleiss, S. Stober, Trustworthy academic risk prediction with explainable boosting machines, in: Proceedings of the International Conference on Artificial Intelligence in Education (AIED), Springer, 2023, pp. 463–475.

[15] D. Joyner, R. Arrison, M. Ruksana, E. Salguero, Z. Wang, B. Wellington, K. Yin, From clusters to content: Using code clustering for course improvement, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE), ACM, 2019, p. 780–786.

[16] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, R. C. Miller, OverCode: Visualizing variation in student solutions to programming problems at scale, ACM Transactions on Computer-Human Interactions 22 (2015).

[17] J. C. Paiva, J. P. Leal, F. Alvaro, Clustering source code from automated assessment of programming assignments, International Journal of Data Science and Analytics (2024).

[18] T. Effenberger, R. Pelánek, Interpretable clustering of students' solutions in introductory programming, in: Proceedings of the International Conference on Artificial Intelligence in Education, Springer, 2021, pp. 101–112.

[19] K. K. Tatsuoka, Rule space: An approach for dealing with misconceptions based on item response theory, Journal of Educational Measurement 20 (1983) 345–354.

[20] A. T. Corbett, J. R. Anderson, Knowledge tracing: Modelling the acquisition of procedural knowledge, User Modelling and User-Adapted Interaction 4 (1995) 253–278.

[21] P. I. Pavlik, H. Cen, K. R. Koedinger, Performance factors analysis –a new alternative to knowledge tracing, in: In Proceedings of the International Conference on Artificial Intelligence in Education (AIED), 2009, p. 531–538.

[22] C. Piech, J. Bassen, J. Huang, S. Ganguli, M. Sahami, L. Guibas, J. Sohl-Dickstein, Deep knowledge tracing, in: Proceedings of the 28th International Conference on Neural Information Processing Systems (NeurIPS), MIT Press, 2015, p. 505–513.

[23] T. Barnes, Novel derivation and application of skill matrices: The Q-matrix method, in: C. Romero, S. Ventura, M. Pechenizkiy, R. Baker (Eds.), Handbook on Educational Data Mining, CRC Press, 2010, pp. 159–172.

[24] M. C. Desmarais, R. Naceur, A matrix factorization method for mapping items to skills and for enhancing expert-based Q-matrices, in: Proceedings of the International Conference on Artificial Intelligence in Education (AIED), Springer, 2013, pp. 441–450.

[25] G. Picones, B. Paassen, I. Koprinska, K. Yacef, Combining domain modelling and student modelling techniques in a single automated pipeline, in: Proceedings of the 15th International Conference on Educational Data Mining (EDM), 2022.

[26] Y. Shi, R. Schmucker, M. Chi, T. Barnes, T. Price, KC-Finder: Automated knowledge componet discovery for programming problems, in: In Proceedings of the 16th International Conference on Educational Data Mining (EDM), 2023.