ART: An Instrument for Developing Algorithmic Reasoning in Programmers

Shruthi Ravikumar
Royal Melbourne Institute of Technology
Melbourne, Australia *s3613612@student.rmit.edu.au*

Margaret Hamilton
Royal Melbourne Institute of Technology
Melbourne, Australia *margaret.hamilton@rmit.edu.au*

Charles Thevathayan
Royal Melbourne Institute of Technology
Melbourne, Australia *charles.thevathayan@rmit.edu.au*

Kashif Ali
Royal Melbourne Institute of Technology
Melbourne, Australia *kashif.ali@rmit.edu.au*

Gayan Wijesinghe
Royal Melbourne Institute of Technology
Melbourne, Australia *gayan.wijesinghe@rmit.edu.au*

Abstract

Despite the significant advances in Information systems adopted for several different courses, the failure rate for Introductory Programming Courses (IPCs) still remains high. At present, the formative activities used in IPCs focus on tracing tasks. However, there is no clear evidence that such tasks foster higher-level abstraction and cognitive reasoning skills needed for code writing. We propose an Algorithmic Reasoning Task system (ARTs) as an instrument, that can be adapted by existing information systems to develop reasoning skills for students learning programming. Our analysis of novice programmer performance reveals that code-writing tasks correlate higher with Algorithmic Reasoning Tasks (ARTs) than with traditional tracing tasks.

Keywords: Algorithmic Reasoning Tasks, Introductory Programming Courses, Teaching programming

1. Introduction

Several Intelligent Tutoring Systems (ITSs) [6], [9], [11], [22], [30] and eLearning platforms [3], [24], [29] have been developed and are adopted for programming courses. Such systems mainly focus on providing automatic feedback to students and navigation support within the course [9]. Instantaneous feedback has shown significant improvement in reducing students' misconceptions [4] and has also reduced manual effort for tutors and lecturers. Automated feedback is provided for several formative activities such as tracing tasks. However, these tasks mainly focus on student comprehension of individual constructs and have done little to develop the reasoning skills needed for code writing. Code writing requires students to develop a unique solution for each problem at hand by composing and interleaving various constructs while reasoning about the overall behaviour of the resulting code. Problems through lack of reasoning skills have been further exacerbated by greater diversity of incoming students, resulting in continued poor performance in programming tests. A survey of 161 institutions worldwide revealed failure rates

in introductory programming course (IPC) is 28% [5]. However, the common code-writing task across four different universities by the McCracken ITiCSE working group confirmed poor coding skill is a widespread and serious concern [18]. Subsequent work using multiple-choice tracing tasks showed a much better performance [14], suggesting they may be measuring different cognitive skills.

In an attempt to find more relevant precursors to coding skills, explain in plain English statements were introduced as part of the BRACElet project with questions set at the relational level of the SOLO taxonomy and the studies have shown "explain in plain English" tasks have a correlation of 0.5586 with code writing skill [16]; novices were classified into different stages based on tracing outcomes. In a more recent work, the higher SOLO level tasks were shown to correlate with improved exam performance [14]. The three lower levels of the SOLO taxonomy focusing on the pre-structural and structural aspects did not require students to abstract the constructs or to distinguish “trees from the forest”[20]. The Neo-Piagetian theory provides the basis for SOLO taxonomy, which suggests novices go through various stages of abstract reasoning [27]. In the initial sensorimotor stage, a novice programmer may not even be able to trace with appropriate values to analyze the behavior of the code. In the preoperational stage, the novice would have developed the ability to generalize the results from different traces to predict the behavior of the code. The novice at concrete operational stage would have the ability to deduct the behavior of the code without having to trace with different values.

The novice programmers who have enrolled in IPCs need early formative feedback to overcome many cognitive conflicts. Students find problem solving difficult as it requires combining higher level thinking, problem abstraction and algorithm development, with language syntax and code tracing [17]. Most tools developed in the past focused mainly on assessing students’ code tracing skills. This paper investigates whether instrument types can be devised that correlate better with algorithmic reasoning and problem abstraction skills needed for problem solving. In the past Parsons puzzle tasks measuring a form of solution planning was shown to correlate better with problem solving than code tracing [10]. In this study we have developed many additional instruments(algorithm detection, algorithm comparison and algorithm analysis tasks) that foster higher order thinking skills necessary for problem solving. The algorithm detection requires in-depth study of an algorithm to extract its overall effect. The algorithm comparison requires identifying different algorithms having the same effect. The algorithm analysis requires reasoning about its behavior for specified criteria such as performance. All of the algorithm-based questions required students to work at the relational level of the SOLO taxonomy.

In this research, we introduce ART as an instrument type that can be widely adopted in on or off-campus e-learning information systems to substantially improve students’ code writing skills and to reduce student failure rates in the course. This research mainly focuses on the following research question RQ. *How well does the performance in ART type questions correlate with for code writing?*

The paper is structured as follow : Section 2 presents the previous work and the gaps while section 3 talks about how the study was conducted including participants details and the different tasks involved in the study. Section 4 and 5 presents the details about how the data was analysed and discusses its findings before drawing some conclusions in the last section.

2. Background

Introductory programming exams commonly exhibit a bimodal distribution in CS1 courses [25],[27]. The “Learning Edge Momentum” (LEM) attempts to explain why high failure rates in programming courses are often combined with a substantially high number of students scoring top grades[27]. Lack of progress in the early stages creates a negative momentum eventually leading to high failure rates while steady progress leads to a positive momentum as new concepts help reinforce earlier foundations. Some have used empirical studies to show how conducting

class tests as early as week 3 can identify students likely to perform poorly in the end of the semester code writing tasks.

Early studies have indicated that there exists a loose hierarchy of skills the students go through while learning IPCs [21]. Some of these studies has also shown the existence of relationship between code tracing, code explaining and code writing and have found that if students are weak at tracing and/or explaining the code then they cannot write systematically[19]. Another similar study conducted at the later stage, highlighted that students misconceptions should be identified at the early stages to make sure their progress in the course in not impeded [19]. One of these studies, investigated on code explaining skills of the students in the end of semester exam states, one should have the ability to the see the forest and not just the trees [20]. In other words, one must possess reasoning skills to be able to write a piece of code. These studies[19, 20, 21, 22] emphasized that, to avoid misconceptions in the students', teachers should consider imparting abstract thinking or reasoning skills to novice programmers.

Parsons puzzle tasks were designed to ease novices into code writing by allowing students to piece together code fragments interactively [10]. Many clues guiding towards the expected solution help students perform better. Spearman ranking coefficient for code writing also showed closer correlation with Parsons when compared to tracing. Parsons however, limits students' freedom in arriving at a solution. Multiple-choice questions (MCQs), when appropriately designed, have been shown to be effective for testing intermediate levels of programming skills. MCQs were found to be the most preferred assessment types in many different domains [12],[16]. Students in general felt such tests can improve their exam performance, as they felt more relaxed [1]. MCQs were also found to be more motivating when used in formative assessments leading to improved self-efficacy when preparing for exams [7].

Several ITS systems [6], [9], [11], [24], [30] developed in the past to teach computer programming also provides MCQs as a formative and summative assessment to effectively teach programming concepts to students. Bayesian intelligent tutoring system (BITS) [6] is one such system developed to provide personalized learning to students using Bayesian Network. A directed acyclic graph (DAG) is constructed for IPC using course textbook. The DAG represents the sequence for learning all the concepts in the problem domain. This DAG is used to provide personalized pathways i.e. navigation support through the course. Students' knowledge for a specific concept in the course is assessed using students' performance to MCQs. However, the MCQs used to assess the student knowledge of each concept does not assess whether the student exhibits algorithmic thinking.

Another research was [11] conducted recently to create a set of instructional strategies that can be used in smart learning systems to increase the student scores, pass rates. This study proposes a predictive model based on certain parameters such as teacher's opinion about students, student's performance to activities such as follow and give instruction, mind mapping activities and lastly gamification. While this predictive model integrated into the ITSs has significant impact on identifying students who are likely to fail, the instructional strategies used do not focus on improving the students' code writing abilities or improving abstract thinking which is essential for students to improve their code writing. The instructional strategies proposed in this research is also based on in-class activities and cannot be widely adopted by all the learning platforms.

Another [30] ITS developed in the past used a very similar approach which does not involve creation of code to assess the students' conceptual knowledge. AtOL [30]is one such web based adaptive ITS developed to assist students in closed laboratories. The system consists of the question tutor, the program tutor and the course management component in which teachers can create new exercises. It includes three instrument types namely true/false questions, MCQs and short-answer questions. The system assesses students preferred choice of programming and question mode and accordingly modify the system to be used in labs. However, it

does not assess the algorithmic reasoning skills of students. Another intelligent interactive educational system ELM-ART [29], consists of five different types of instrument namely yes or no questions, forced- choice question items, MCQs, free-form question types and lastly fill the gap question types which are situated within the "electronic textbook" and are automatically marked. The performance of students to these instrument types in each page of the textbook determines whether the student can navigate to next page of the book in the course. The system provides adaptive feedback and navigation support based on student's performance to different instrument types.

Similarly CIMEL ITS [24], uses quiz questions and interactive exercises like drag and drop activities to measure the students' conceptual knowledge. It relies on Bayesian network-based domain model to predict whether the student understands the specific concept based on their performance to the exercises given by the ITS. Similarly, there are several e-learning platforms [2],[13],[22] developed to aid lecturers and students to manage the course content. E-learning platforms have enhanced the teaching of programming courses by providing a centralized platform for students' and lecturers. It provides the ability to access course resources, seek feedback from lecturers on different programming exercises, attempt quizzes on different modules of course. CPS [2] is another e-learning system developed to teach python programming. It has offered a scalable platform using modular micro-service oriented architecture which combines features and integrations from Node.js, React, Python, Nvidia Docker, Jupyter etc. to accommodate usage of the system by multiple users concurrently without affecting the performance. The system has been well received by teachers and the students. A very recent study [8] to analyze the student completion rate of MooCs used the student's performance in weekly quizzes to determine the student behavior. It has shown that students who complete the quizzes are likely to complete the course compared to others. It is clear from this study that assessments used during the course plays an important factor to determine student completion and success rates and it is necessary to have right instrument types in any course.

While these information systems developed to aid novice programmers and lecturers in IPCs assists in resolving student difficulties and provide personalized pathways, we contend there is a need to come up with instrument types that are capable of imparting and assessing reasoning skills needed for code writing and to reduce the failure rate in programming courses.

3. Methodology

To correlate the performance in problem solving with different objective type questions, a 110-minute test was conducted which included 8 objective questions and 3 code writing tasks. The code writing questions were designed to assess whether a student can abstract a problem by translating it from the problem domain, come up with a viable algorithm and implement it using the language constructs. The questions selected for this study were different from any they had come across during the semester. Moreover, the students were not given any sample practice questions to avoid any external influence. Students were advised to spend about 30 minutes on the objective questions (Tracing and Algorithmic Reasoning Tasks) and the remaining 80 minutes on code-writing. The ARTs questions introduced in this study were not just an ordinary objective question but it was designed in such a way where the students need to understand the overall purpose of the code or in other words exhibit abstraction skills to answer these questions (Refer Table 1 to understand the different ART Type questions used in the study and their purpose). The data was collected from the Introduction to Programming course students in the penultimate week on 2019 semester 1, allowing questions covering all the topics taught in that semester. The data was collected electronically through google forms and there were 56 participants in the test. The objective questions were automatically marked while code writing tasks were manually marked. A positive marking scheme was used [8] with each component awarded 0.5 or 1 mark depending on whether it is partially or fully correct. However, no partial

Q: What will be the output of the program below?

```
public class TL {
   public static void main (String[]
args) {
      int T1 = 1;
      int T2 = 1;
      int T3 = 0;
      for (int i=1; i<=4; i++) {
            T3 = T1 + T2;
            T1 = T2;
            T2 = T3;
      }
      System.out.println(T3);
   }
}
```

Fig. 1. Tracing Question Sample

marking is given for ARTs and tracing tasks.Spearman rank coefficient was used as the primary means of computing correlation between ART questions and code-writing.

3.1. Algorithmic Reasoning Tasks

Our prior experience devising, administering, and analysing weekly quizzes revealed detect-and-apply-algorithm tasks showed consistently greater discrimination index when compared to conventional tracing tasks. Such tasks required students to detect the purpose of the algorithm before applying it to 6 to 8 different inputs. While it is not possible to prevent students to manually trace the algorithm several times to work out the final output, the time restriction made such an approach viable. In addition to algorithms detection we also introduced new types of objective questions requiring students to focus on the structural aspects of algorithms as shown in Table 1.

Table 1. Types of Algorithmic Reasoning Tasks

Purpose	***ART Type***
Detection	Requires abstraction skills to detect what the role of the algorithm. Students are expected to apply cognitive skills at relational level to analyze how the behavior will change for different inputs.
Comparison	Students are expected to identify algorithms which will display the same collective or composite behavior considering different input values.
Analysis	Students are expected to analyze an algorithm including working out worst case scenarios considering all possible paths.

The sample questions for each task type namely Tracing and ARTs tasks are provided in Fig.1,2,3 and 4 respectively.

3.2. Code Writing Tasks

The code writing tasks were primarily designed to measure problem analysis, solution planning, coding, and desk checking. One of the questions (refer Fig.5) which we used in our analysis was designed to test whether students can abstract a familiar problem and design an algorithm before implementing it in Java.

4. Analysis of Marks Distribution and Correlation

56 students participated in this study and their answers to both ARTs and code writing tasks are analysed here. The code writing marks were mainly allocated for analyzing the problem, coming up with a viable strategy and coding the algorithm in Java. No marks were deducted for minor syntax errors. There were 2 tracing tasks and 6 instances of ARTs questions: 2 in

Q: Assume the following program segment is run several times with different values stored in the int array x as shown below. What will be the output (count2) for different values of x?

```
int count1 = 1;
int count2 = 1;
int x[] = …
for (int i=1; i< x.length; i++) {
   if (x[i] == x[i-1])
      count1++;
   else
      count1 = 1;
   if (count1 > count2)
      count2 = count1;
}
System.out.println(count2);
```

x[]	count2
int x[] = {10,2,4,4,8,3,3,3};	
int x[] = {10,13,13,2,2,2,10};	
int x[] = {9,2,4,7,8,5,9};	
int x[] = {7,2,4,9,4,2,7};	
int x[] = {5,4,9,8,10,8,9,8};	
int x[] = {5,2,8,9,9,8,2,5};	

Fig. 2. Art-Detection Question Sample

Q: Which of the following 3 program segments using an array of 3 int elements are equivalent (will always have the same outcomes)?

```
a[2] += a[1];     a[1] += a[0];     // I
a[2] += a[0];     a[2] += a[1];     // II
a[1] += a[0];     a[2] += a[1];     // III
```

Fig. 3. ART-Comparison Question Sample

Q: Assume the following program segment is run several times with different values assigned to 'v'. What will be the maximum possible value printed (for count)?

```
int a[] = {3,5,8,13,16,19,23,26,36,67,69,70,80,89,90,98};
int start = 0;
int end = a.length - 1;
int count = 0;
int index = -1;
int v = …;  //  can be any integer

while (start <= end) {
   count++;
   int mid = (start + end )/ 2;
   if ( a[mid] > v ) end = mid - 1;
   else if ( a[mid] < v ) start = mid + 1;
   else { index = mid; break; }
}
System.out.println(count)
```

Fig. 4. ART-Analysis Question Sample

Q: Write a program to determine whether given 3 sides can or cannot form a triangle. For a valid triangle any one side should be smaller than the sum of the other two sides.

Fig. 5. Code Writing Question Sample

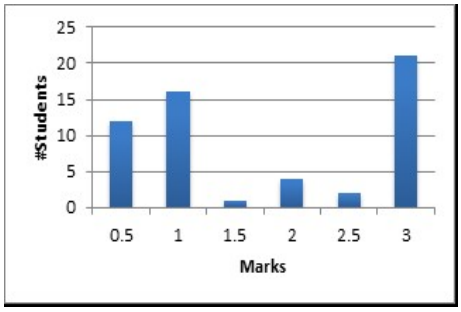

Fig. 6. Distribution of Marks in Code Writing

algorithm comparison, 2 in algorithm detection and 2 in algorithm analysis and 3 code writing tasks.

4.1. Distribution of Marks

This section presents the distribution of marks for various ARTs and code writing sections before correlating their performance. Fig.6 depicts the distribution of marks for the code writing task revealing a classic bi-modal distribution with 40% of the students scoring either 0.5 or 1 out of 3 and another 40% scoring 3 out of 3. Fig.7 shows the distribution of marks for the 3 ARTs algorithm comparison, detection, and analysis & tracing. Note, the ART analysis type question has the most number of students getting 0 out of 3, and tracing the least. Fig.7 shows a summary for the 4 task types with two tasks in each type, in terms of the difficulty index, discrimination index and standard deviation. In all ARTs no partial marking is given (unlike code writing), with students getting either 0 or 1. Difficulty index is the proportion of students answering a question correctly. Hence more difficult items have low values for difficulty index. Discrimination index distinguishes how an item distinguishes those with higher skills from those with low skills. We used the top third and bottom third overall scores (including objective and code writing) to arrive at the discrimination index. The discrimination index and difficulty indices in Table 2 vary substantially between different ARTs types. These indices can also vary within the task type depending on familiarity, algorithmic complexity, level of abstraction needed and whether

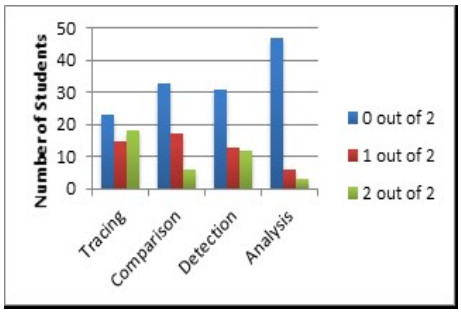

Fig. 7. Distribution of Marks in ARTs & Tracing

Q: Which of the following 3 program segments using 3 integer variables (assigned 1 to 100) will have the same outcomes?

```
x = x + y; y = x - y; x = x - y; // I
x = x - y; y = x + y; x = y - x; // II
x = x + y; x = x - y; y = x - y;  // III
```

A) I and II B) I and III
C) II and III D) I, II and III
E) None of these segments have the same outcome

Fig. 8. Algorithm-Comparison Task Sample

Table 2. Difficulty,Discrimination Indices & Std. Deviation

Statistics	Non-ARTs	ARTs		
	Trace	Compare	Detect	Analysis
Difficulty Index	0.46	0.26	0.33	0.11
Discrimination Index	0.74	0.47	0.66	0.29
Standard Deviation	0.86	0.69	0.82	0.53

any clues (lead-in) are given. In the algorithm comparison task for example, the code shown below (refer Fig. 8)had much lower discrimination index than others and therefore excluded in the analysis. Though the first two fragments in this task are simply swapping the variables, the lack of clues, lack of familiarity and the need for a higher level of abstraction may have led to the weak performance in this task.

4.2. Correlation with Code writing

We measured the correlation using the Spearman rank coefficient for tied ranks as the limited range for ARTs led to many tied ranks. Others have used such a coefficient to correlate Parsons with code writing and tracing [8]. Table 3 shows the correlation between different parts of ARTs and code writing, with algorithm detection showing the highest correlation. The low correlation for algorithm comparison is partly due to the use of multiple-choice questions where random selection can distort the statistics. The low-ranking correlation is used for analysis because over 80% of the students got 0 out of 3 for analysis, leading to many tied ranks. Since algorithm detection and tracing show the highest correlation in Table3, these are further analyzed in Fig.9, which show the distribution of students' scores between these categories and code comparison respectively. The area of each circle represents the number of students with the scores in that intersection. The left side of Fig.9, suggests high marks in code writing is not predicated on getting high marks in tracing; students getting low marks in tracing have clusters where they score both low and high marks in code writing. This section therefore computes the conditional

Table 3. Spearman Rank Correlation with Code Writing

Non-ARTs	ARTs			
Tracing	Comparison	Detection	Analysis	ARTs
0.42	0.36	0.61	0.36	0.44

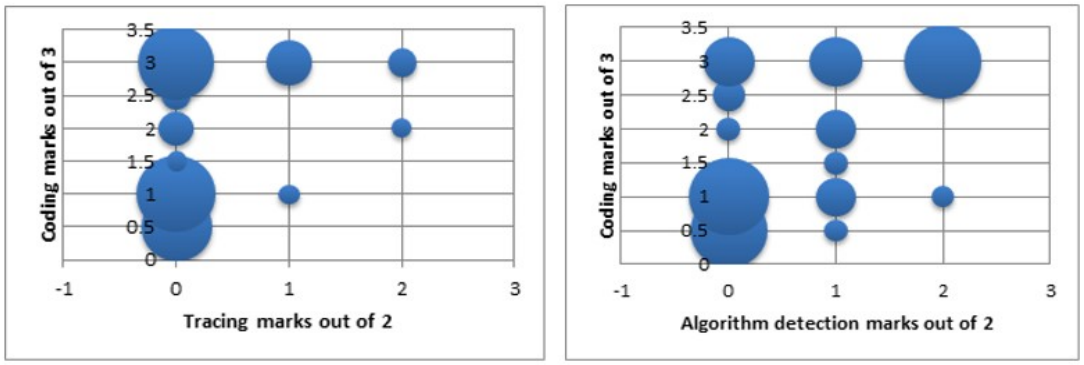

Fig. 9. Correlation Tracing Vs Coding (Left) and Correlation Algorithm Detection Vs Coding (Right)

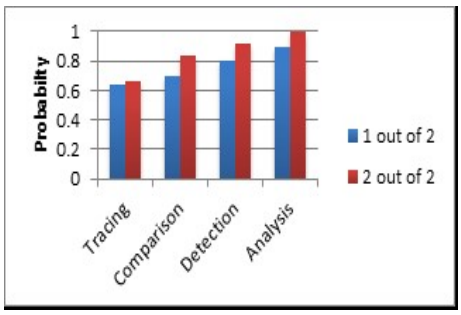

Fig. 10. Conditional Probability of Passing Code Writing

probability of passing code writing given their performance in different ARTs. Fig.10 shows the conditional probability of a student with 1 or 2 marks in objective section getting code writing correct are substantially higher for algorithm reasoning tasks than tracing.

5. Quantitative Analysis and Discussion

This study has helped us identify the algorithmic reasoning tasks that can be incorporated in the weekly quizzes to impart algorithmic reasoning skills in students and improve their code writing skills.

The high correlations between code writing and algorithmic reasoning tasks in Fig.10, suggests structural and relational thinking needed for algorithmic tasks share many of the reasoning skills needed in code writing. For example, ability to compare algorithms for their structural properties is predicated on being able to arrive at derived behaviors of various composed elements focusing on their relational aspects. Identifying the derived behavior of an unfamiliar algorithm (such as locating an element using binary search) is also essential when asked to apply it on many different inputs, as line-to-line tracing becomes nonviable. Similarly analyzing algorithms for their structural properties fosters reasoning skills in students.

Table 4. Conditional Probability of a Student Failing Tracing but Passing ARTs

ARTs		
Comparison	Detection	Analysis
0.26	0.17	0.0

Both Spearman rank coefficient and conditional probabilities linking various tasks to code writing in Table 3 and Fig.10 suggest algorithm detection activate higher cognitive skills needed for problem solving than tracing. Moreover, the algorithms students are asked to detect and apply are unfamiliar to novice programmers. Having to write down the answers when an algorithm is applied to numerous inputs makes it difficult to trace over and over again when time is limited. The essence of these tasks is therefore providing a way to demonstrate what the code does by applying it to different inputs, instead of explaining code in words that requires manual marking.

Spearman rank coefficient had slightly lower value for algorithm comparison than tracing when correlating with code writing. However, these differences can be attributed to the guessing factor present in multiple-choice questions (5 choices) with 0.2 probability of getting the question correct with random guessing. We therefore argue the actual correlation with code writing is likely to be much higher if guessing can be prevented or reduced through negative marking. Moreover, left side of Fig.9 reveals most students scoring high marks have fared poorly in tracing, affirming earlier findings [18]. Use of conditional probabilities in Fig.10 also shows algorithm comparison tasks correlate better (0.84) with coding than tracing (0.66).

Spearman rank coefficient also had slightly lower value for algorithm analysis tasks than tracing when correlating with code writing. The main reason was analysis tasks answered correctly by only 11% of students was a much smaller group compared to others, thus making the ranking less reliable. However, Fig.10 reveals those scoring well in algorithm analysis tasks have a much higher conditional probability of passing code writing, suggesting analysis tasks demand the same cognitive skills needed for problem solving. Fig.10 depicting conditional probability of passing code writing as a function of passing other types of tasks shows it is least dependent on tracing skills. However, Table 4 suggests tracing is a precursor skill for algorithmic reasoning tasks, as students unable to pass tracing have very low probability of passing ARTs. It is evident form Fig.10, code writing correlates increasingly higher with algorithm comparison, algorithm detection and algorithm analysis tasks in that order, when compared to tracing, thus answering our research question RQ.

Algorithm analysis performing various what if scenarios to arrive at the best or worst-case scenarios appears to need more in-depth reasoning than algorithm detection and comparison aggregating various parts to arrive at the emerging behavior. Code detection verifies the behavior by getting students to apply it on specific inputs instead of expressing it in words. Algorithm comparison requires matching equivalent algorithms while algorithm efficiency requires reasoning to arrive at the worst or best performances. Thus, all these ARTs can be presented as fill in the blank or multiple-choice questions, which can be easily automated. We have used only three types of ARTs questions in our study. A human expert (educator) will likely use additional types of questions. These ARTs question types can be widely adopted in any e-learning platforms or ITSs to impart algorithmic reasoning skills and to predict student navigation pattern or student success in course using these task types. In future work we intend to develop other instrument types, introduce these instruments as part of weekly quizzes and measure the improvement in student performance.

6. Conclusion

Our study revealed ARTs have the potential to be used as an alternative to some code-writing tasks, in formative assessments and for self-learning. As the marking of ARTs can be easily adopted by existing information systems in the form of multiple-choice questions, more frequent assessment is possible compared to code-writing. Unlike code-explain tasks used in assessing student cognitive development levels, they do not require instructor involvement and therefore are scalable. The three algorithmic reasoning tasks we devised showed substantially different difficulty and discrimination indices, permitting an incremental approach to fostering problem

solving skills in large diverse student cohorts. The novel algorithm detection task is similar to the code-explain strategy, as it requires students not only to extract the algorithm but also to apply it to at least 6 different inputs, thus emphasizing the relational aspects. The algorithm detection and algorithm analysis tasks showed the strongest conditional probabilities of getting code-writing correct suggesting these tasks exercise similar cognitive skills as code writing. Our findings suggest students in large diverse introductory programming classes can improve their coding skills by combining tracing and algorithmic reasoning tasks in formative and practise activities.

References

1. Abreu, P.H., D.C. Silva, and A. Gomes, Multiple-Choice Questions in Programming Courses: Can We Use Them and Are Students Motivated by Them? ACM Transactions on Computing Education (TOCE), 2018. 19(1): p. 1-16.
2. Alexandru, D., A. Iftene, and D. Gîfu, Using New Technologies to Learn Programming Languages. 2019.
3. Anderson, N. and T. Gegg-Harrison. Learning computer science in the" comfort zone of proximal development". in Proceeding of the 44th ACM technical symposium on Computer science education. 2013.
4. Barra, E., et al., Automated Assessment in Programming Courses: A Case Study during the COVID-19 Era. Sustainability, 2020. 12(18): p. 7451.
5. Bennedsen, J. and M.E. Caspersen, Failure rates in introductory programming: 12 years later. ACM inroads, 2019. 10(2): p. 30-36.
6. Butz, C.J., S. Hua, and R.B. Maguire. A web-based intelligent tutoring system for computer programming. in IEEE/WIC/ACM International Conference on Web Intelligence (WI'04). 2004. IEEE.
7. Clark, D., Testing programming skills with multiple choice questions. Informatics in Education-An International Journal, 2004. 3(2): p. 161-178.
8. Cristea, A.I., et al., How is Learning Fluctuating? FutureLearn MOOCs Fine-Grained Temporal Analysis and Feedback to Teachers. 2018.
9. Crow, T., A. Luxton-Reilly, and B. Wuensche. Intelligent tutoring systems for programming education: a systematic review. in Proceedings of the 20th Australasian Computing Education Conference. 2018.
10. Denny, P., A. Luxton-Reilly, and B. Simon. Evaluating a new exam question: Parsons problems. in Proceedings of the fourth international workshop on computing education research. 2008.
11. Figueiredo, J. and F.J. García-Peñalvo. Intelligent Tutoring Systems approach to Introductory Programming Courses. in Eighth International Conference on Technological Ecosystems for Enhancing Multiculturality. 2020.
12. Furnham, A., M. Batey, and N. Martin, How would you like to be evaluated? The correlates of students' preferences for assessment methods. Personality and Individual Differences, 2011. 50(2): p. 259-263.
13. Huang, C.-J., et al., Developing an intelligent diagnosis and assessment e-learning tool for introductory programming. Journal of Educational Technology Society, 2008. 11(4): p. 139-157.
14. Izu, C., A. Weerasinghe, and C. Pope. 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. 2016.
15. Kasto, N. and J. Whalley. Measuring the difficulty of code comprehension tasks using software metrics. in Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136. 2013.

16. Kuechler, W.L. and M.G. Simkin, How well do multiple choice tests evaluate student understanding in computer programming classes? Journal of Information Systems Education, 2003. 14(4): p. 389.
17. Lin, J.M.-C., K.-Y. Lin, and C.-C. Wu, A content analysis of programming examples in high school computer textbooks in taiwan. Journal of Computers in Mathematics and Science Teaching, 1999. 18(3): p. 225-244.
18. Lister, R., et al., A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bulletin, 2004. 36(4): p. 119-150.
19. Lister, R., C. Fidge, and D. Teague, Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. Acm sigcse bulletin, 2009. 41(3): p. 161-165.
20. Lister, R., et al., Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. ACM SIGCSE Bulletin, 2006. 38(3): p. 118-122.
21. Lopez, M., et al. Relationships between reading, tracing and writing skills in introductory programming. in Proceedings of the fourth international workshop on computing education research. 2008.
22. Malik, S.I., et al., Learning problem solving skills: Comparison of E-Learning and M-Learning in an introductory programming course. Education and Information Technologies, 2019. 24(5): p. 2779-2796.
23. McCracken, M., et al., A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, in Working group reports from ITiCSE on Innovation and technology in computer science education. 2001. p. 125-180.
24. Moritz, S.H., et al., A" design-first" curriculum and Eclipse™ tools. Journal of Computing Sciences in Colleges, 2007. 22(3): p. 51-52.
25. Robins, A., Learning edge momentum: A new account of outcomes in CS1. Computer Science Education, 2010. 20(1): p. 37-71.
26. Shuhidan, S., M. Hamilton, and D. D'Souza. A taxonomic study of novice programming summative assessment. in Proceedings of the Eleventh Australasian Conference on Computing Education-Volume 95. 2009. Citeseer.
27. Teague, D., et al. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. in Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]. 2013. Australian Computer Society.
28. Venables, A., G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. in Proceedings of the fifth international workshop on Computing education research workshop. 2009.
29. Weber, G. and P. Brusilovsky, ELM-ART: An adaptive versatile system for Web-based instruction. International Journal of Artificial Intelligence in Education (IJAIED), 2001. 12: p. 351-384.
30. Yoo, J., et al. Intelligent tutoring system for CS-I and II laboratory. in Proceedings of the 44th annual Southeast regional conference. 2006.