Stochastic Tree-Based Generation of Program-Tracing Practice Questions

Anderson Thomas
The George Washington University
Washington, DC
athomas1@gwmail.gwu.edu

Troy Stopera
The George Washington University
Washington, DC
troystopera@gwmail.gwu.edu

Pablo Frank-Bolton
The George Washington University
Washington, DC
pfrank@gwmail.gwu.edu

Rahul Simha
The George Washington University
Washington, DC
simha@gwmail.gwu.edu

ABSTRACT

Recent work [6, 23] has shown that mental program-execution exercises, in the form of Parson's puzzles or program-tracing, are effective in improving student performance in intro CS courses. This form of practice is promising because its low cost of creation and short duration (for the student) can promote the significant practice needed for learning. The goal of this paper is to enable wider use of such exercises through large-scale automated generation of short, multiple-choice mental execution questions. The challenge in automation is to algorithmically generate effective *distractors* (plausible, but incorrect choices), and to generate questions of varying levels of difficulty and whose difficulty level can be set by the instructor. In this paper, we propose a language-generalizable approach for automatically generating a practically unlimited number of such exercises, each constructed to a designated level of difficulty and incorporating the core programming-in-the-small themes: assignment, conditionals, loops, and arrays. The stochastic tree-based generation algorithm and a subsequent simulation of execution also enable generating effective distractors since all possible execution paths are readily available in the tree at the time of generation, and the distractors, therefore, correspond to reasonable (but ultimately incorrect) paths of execution. Furthermore, the approach is easily transferable to other languages with little effort. The generated questions are delivered through a mobile app that can be customized by the instructor to vary the questions generated and to introduce interleaving to take advantage of the spacing effect. Preliminary student feedback on the experience has been positive.

KEYWORDS

Intro CS, Question Generation, Practice Tools

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287492

ACM Reference Format:
Anderson Thomas, Troy Stopera, Pablo Frank-Bolton, and Rahul Simha. 2019. **S**tochastic Tree-Based Generation of Program-Tracing Practice Questions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287324.3287492

1 INTRODUCTION

The past decade has seen a surge of interest in enhancing student learning in introductory programming courses with activities that go beyond the traditional lecture followed by programming assignment [9, 10, 19]. One such enhancement, the use of program-tracing exercises, has been correlated to an improvement in the quality of programming skill in students [6, 18, 19, 23]. Systematically offering students many small exercises, in various gradations of difficulty and repeatedly, also aligns well with research from the science of learning, for example, *deliberate practice* [7], in which exercises are customized to the individual learner's level, provide immediate feedback, and are in the learner's *zone of proximal development* – just slightly harder than what the learner is capable of. An ideal support system for deliberate practice in programming, in addition to regular programming assignments, should also allow the instructor to set some broad parameters (numbers of questions, mix of difficulty levels), should be efficient in generation to enable scaling with numbers of students, and should reach students where they are (on their smartphones) to encourage repeated practice. These requirements pose a problem for hand-crafting custom questions, already a significant time sink for instructors, who might opt-out of these approaches to avoid the additional overhead.

One way to generate lots of questions is to use *question templates* that let an instructor specify which parts of a question can be randomly generated or selected, which then enables automated grading [3, 4, 8, 10, 11, 14, 17, 23]. The human effort involved in template-generated questions is both its advantage and disadvantage. Instructors thoughtfully generate and identify opportunities for learning through careful crafting of templates, expertly judge the level of difficulty, and are even able to include appropriate hints if the generated question is incorrectly answered. At the same time, the writing of templates takes time, as does vetting the generated questions, and ensuring that at least some distractors in each question are meaningful.

The goal of this paper is to ask whether a more automated approach – ours is called GenCODE – can be effective. In particular, can an algorithm judge the difficulty level during generation, provide sufficient variety, and most importantly, create meaningful distractors? We propose a stochastic tree-based approach that works in two phases, first to generate the code for a tracing exercise, and second, to generate the distractors for multiple-choice questions. (The latter is less important in variations where students must fill code.) The first phase mirrors a recursive-descent parser in that a tree similar to a parse-tree is constructed by randomly expanding nodes representing statements into assignments, loops, conditionals – some of which have statement-blocks that are then recursively expanded. The second phase simulates the execution of the generated tree to explore a variety of execution paths with the intent of identifying incorrect distractors which might appear plausible to students.

The GenCODE system itself consists of an instructor interface that gives the instructor the ability to set broad parameters. The generated questions are then available to the instructor for further editing or vetting, if the instructor so chooses. The questions are then delivered to a smartphone app, where they are gamified into various levels to encourage students to win points. As each student starts to answer questions and "level up" in the game, they are given harder questions. Furthermore, when a student incorrectly answers a question, the system will ensure that the question is presented back to the student *at a later time*, to exploit the learning that arises from the *spacing effect* [2].

GenCODE was piloted in a brief two-week trial in an introductory semester-long programming course (in Java) at our university during the Fall of 2017 and Spring 2018. Our main findings in this experience-report are the following. First, when human experts evaluate the difficulty level of generated questions, the human rating of difficulty has reasonably close correlation to the algorithm's rating of difficulty, suggesting that the approach has the potential for producing questions in the the desired student's zone of proximal development. Second, students were found to often get questions wrong and get the same questions wrong multiple times, which suggests that the algorithmically produced distractors are reasonable.

Third, students surveyed about their experience responded positively, appreciating the opportunity for low-stakes practice. Finally, using a Turing test, we attempted to discern whether experts (seniors, grads) could distinguish between the algorithm's generated questions and similar hand-crafted questions from other experts. Here, we discovered that our approach needs refinement: human experts were able to intuit some inexplicable signature from the algorithm-generated questions. Accordingly, some surface features of the tree-generated code may need modification; however, the questions are nonetheless useful for practice.

In the following sections we describe the related work on question generation and the implementation details of our system, followed by a discussion of the types of questions involved and the response to the system.

2 RELATED WORK

Computer Science educational research has investigated a variety of approaches to support the learning of programming that go beyond the typical weekly programming exercises, including for example, pair-programming, live coding and gamification. Program-tracing, the focus of this paper, is one such approach that researchers have found effective [16, 18, 19]. For example, Kumar et al [16], used a controlled study to show that code-tracing can improve code-writing abilities of students. A concomitant growth in delivery platforms has accompanied the interest in types of exercises: many researchers have elected to build systems designed to work on mobile devices [5, 12, 20]. This has allowed researchers to create program-assembly questions based on predefined code snippets. These findings have led to the integration of programming questions into new mediums, as seen in the work of Oyalere et al [20], where gamification is applied to Parsons Puzzles. Deb et al.[5] extend these tracing tasks with the inclusion of per-question feedback. More recently, Ericson et al. [6] found that program-question generation with automated feedback provides a system for reducing the time-cost of practicing programming through tracing activities, while yielding the equivalent results to activities that focus on writing code. Particular focus has surrounded Parson's Programming Puzzles[21], which are puzzle-like questions where students complete programs by correctly ordering code snippets.

In related work, the description of a program's execution as a method of explaining the correct answer has been shown to yield improved program writing skills in students [4, 8]. Focusing on the specific domain of *counter-controlled loops*, Kumar [4] presents a template-based system for generating C++ code snippets and questions. In [22], Prados et al. obtain some reuse of their system by changing the programming language of the generated code snippets.

Brusilovsky et al. focused on the development of tools for out-of-classroom self-assessment [3]. In their study, students answered sets of template-generated questions, resulting in the detection of a significant improvement of student understanding of program semantics. Variability in the generated material was obtained through the use of randomized parameters within highly structured code snippets [3]. With respect to the domain of *variable scope*, Fernandes et al. [8] found improved learning when studying the coupling of a question generator and a feedback system. In the work of Kumar at al. [14] on question generation, and specifically on domain-tree models capable of describing scope, the authors describe a system notable for being capable of providing feedback at various conceptual levels. In addition, the authors found improved learning when studying the coupling of a question generator and a feedback system. The work of Hsiao et al [10] describes in detail QuizJET, a templating system for object-oriented programming. QuizJET was designed to focus on a narrow subset of programming concepts and required on the order of 100 handcrafted templates for generating questions. These studies showcase the ability of templating systems to cover and provide materials for new topic domains but simultaneously illustrate the difficulty of extending them to new topics. With a different approach, Zavala et al. [23] move away from the strict templating systems and use ontological elements to provide randomized questions. Through the use of linked open data, Zavala

et al. were able to add relevance to questions from real data sources. This approach, as well as the one we describe in this paper, lends itself to a more scalable paradigm for question generation. Related to this, we note that others have focused on providing feedback or tutoring specialized topics [1, 3, 8, 11–13, 15].

In contrast, our work is primarily aimed at simplifying and automating the *generation* of program-tracing questions by exploring whether structural (tree-based) algorithms have the potential of generating useful questions.

3 THE GENCODE ALGORITHM AND SYSTEM

In the following sections, we describe the stochastic tree-based approach to the generation of multiple-choice program-tracing questions, and the methods used to obtain variability in structure and difficulty. GenCODE is itself written in Java and currently generates Java questions, although the approach is easily applied to other languages.

3.1 Generating Program Structure

We start by describing the first step in generating a question: the generation of the code on which the tracing question is based. At the current time, GenCODE is capable of generating programs on topics such as *assignments*, *for-loops*, *conditionals*, and *for-arrays*. These, in turn may be combined (sequentially or through nesting) to create problems of increased complexity. The instructor indicates the topics of interest, with a broad difficulty category (easy, medium, hard) for each topic, which GenCODE then uses to generate questions.

As mentioned earlier, the general principle used is to generate recursively a tree that resembles a parse-tree by starting with a root node (a code block), which then has randomly generated child nodes (statements), each of which can be recursively expanded at random. However, solely using a language grammar to randomly generate terminal strings (programs, in this case) can result in strange-looking and unrealistic code, or sometimes, code that does not execute, and thereafter creating the burden of vetting on the part of the instructor. To constrain this process and generate valid code useful for tracing questions, we use three strategies with the goal of creating program-tracing questions that are restricted by parameters selected by the instructor, guaranteed to execute, and useful in bringing out common student misconceptions.

The first strategy is to present the instructor with an interface to specify topic combinations. These options are interpreted from the topics indicated by the instructor to the form of *patterns*. A pattern is the desired outline of the target program, along with a limited nesting depth. Currently, GenCODE has implemented seven patterns encompassing the following basic intro-CS topics: *for-loops*, *conditionals*, and *for-arrays*. The patterns are:

- Nested Patterns: These combine building blocks (NestedLoop, NestedConditional, and NetstedLoopConditional)
- Non-Nested Patterns: These provide a non-nested block (SingleLoop, SingleConditional, and ComboLoopConditional)
- Manipulation patterns: These manipulate some part of the code to create *distractor* answers: (ArrayWalk and LoopSkipManip)

The hope is that, through a quick glance at the name, an instructor can intuit the type of code that will be generated. At this time, any of these may be freely combined, except for *conditionals* and *for-arrays* (which remains a work in progress).

Note that patterns differ from templates in the amount and type of information that is predefined. While templates usually fix structure and alter coefficients and constants, our patterns are simple outlines of topic combinations that can be recursively expanded. The actual tree depth of any generated code can vary because the elements of the tree are randomly generated.

The second strategy is to limit the depth because, if a generated program is too long and unwieldy, program tracing becomes too difficult for intro-CS students.

The third strategy is to build the tree not out of traditional grammar elements but out of larger units called *blocks* that better align with the goals of identifying opportunities for learning. For example, an *evaluation block* can be a boolean expression (that itself may contain arithmetic expression) built out of variables declared prior to the block. Such a block can be used in a conditional or in a for-loop. What makes this higher-level structure useful is that the handling of an alternative execution pathway from an evaluation block is similar regardless of where it is used. In this manner, we also define other blocks such as *statement blocks*, and *component blocks*. A statement-block is equivalent to a leaf node in the tree, consisting of simple variable declaration or assignment, while a *component block* can be instantiated, for example, as a for-loop (which itself has a component block and evaluation block). Lastly, because tracing questions most often involve program output, a component block may include a print statement.

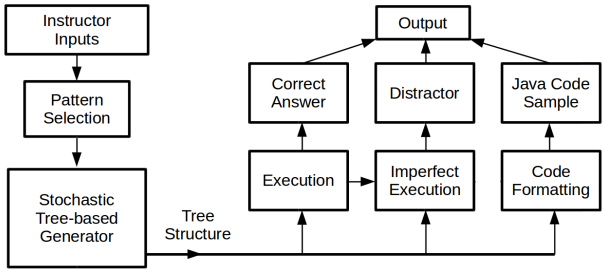


Figure 1: GenCODE structure

Figure 1 shows a high-level overview of the components in the system. The instructor specifies generation parameters (patterns and difficulty levels), and the tree-based algorithm generates a code snippet in the form of a tree. The tree is then explored for execution pathways, as explained below, to generate the correct answer and distractors that are clearly different yet plausible.

Two example code snippets are shown below. Both were generated by GenCODE when given the input topics "for-loops and for-arrays", and difficulty setting of 0.1 (easy), and 0.9 (hard), respectively.

```
public int easy_example(){
  int[] a = {18, 11, 37, 33, 27, 21};
  int b = 99;
  int c = 97;

  for (int c=0; c<a.length; c = c+1){
    a[c] = c;
    b = b*c;
  }
  return a[5];
}
```

```
public int hard_example(){
  int[] a = {46, 21, 62, 75, 96, 89, 84, 92, 51, 85};
  int b = 47;
  int c = 97;
  int[] d = {33, 46, 72, 2, 81, 84, 18, 89, 66};
  int e = 96;

  for (int f=0; f<d.length; f = f+1){
    for (int g=0; g<10; g = g+2){
      for (int h=5; h>=2; h = h-2){
        d[f] = g;
        e = e*h;
      }
    }
  }
  return d[5];
}
```

For the *easy* case, the correct choice is *5*, with distractors of: *8, 6,* and *3*. For the *hard* case, the correct choice is *8*, with distractors of: *9, 6*, and *11*. It is a simple matter to modify the difficulty settings inside GenCODE to narrow or widen the difference between extreme difficulty settings. For the pilot where the system was tested, the settings were adjusted to a more "narrow" spread so that even the "hard" problems are solvable by novice programmers.

3.2 Virtual Execution of Generated Code

Rather than generating code that is then compiled for actual execution, we simulate the execution within the tree itself. This is useful because the tree is a convenient structure that embodies all possible execution paths, some of which serve as candidates for producing distractors, and it is a convenient code data-structure for tracking variables and making adjustments.

The execution simulation tracks variables and the values in them, because ultimately, the values of some variables are printed (or returned) and are therefore ideal for generating "What does this method print (or return)?" questions. The simulation engine is independent of the generation, and can simulate any combination of topics and depth when given a tree.

Distractors, in a multiple-choice setting, are possible answers that vary in their similarity to the correct option. Good distractors are plausible answers that cause the student to pause and think carefully. Generally, the more plausible the distractors, the harder the question. During the simulation, the algorithm can change the path of execution at will, sometimes stopping for a partial answer (the current state of the variable to be printed), or following an alternative (but ultimately wrong) path of execution. For example, one distractor could offer an answer choice that results from having stopped a loop before its completion. Another might obtain a faulty final state by skipping an iteration step. The resulting final value of the incomplete or modified execution can be used as a distractor answer.

One important aspect of this procedure is that the choosing of each type of erroneous option (incomplete execution, skipped iterations, inverse logic, etc) points to specific conceptual errors that the student might be consistently falling prey to. This might give an insight into conceptual gaps that the student may have and help craft corrective actions (by designing specific sets of questions) that precisely target them.

3.3 Question Difficulty

One of the key principles of *deliberate practice*, the well-studied theory of how practice needs to be structured [7], is that practice exercises need to be in the learner's zone of proximal development: just slightly harder than what their current ability can tackle. For automated generation, this translates into the problem of generating questions with a target level of difficulty so that, as students gain in performance, the system presents with increasingly harder questions. This goal is challenging both because question hardness is subjective and we know little about the connection between program structure and the difficulty of resulting questions about tracing the program.

Our approach for this aspect of the system is to start with something heuristic, and to test the results with actual users assessing the level of difficulty. In this way, if there is reasonable correlation, the heuristic approach can be used to generate questions with a desired level of difficulty in order to support optimal practice. For the heuristic, we loosely use a combination of the program elements and its length as a measure, and successively build a piece of code that meets the target level of complexity, which we use as a proxy for the level of question difficulty. The instructor chooses a difficulty level between 0 (easiest) and 1 (hardest). Then, to generate a question with the requisite difficulty, the algorithm first randomly chooses a pattern and then uses the recursive generation approach to extend the complexity of the code to achieve the target difficulty level. The actual assessment of individual component difficulty is heuristic, and initially devised through some trial and error. In the next section, we describe the results of some user testing that supports our general approach although further refinement is needed for higher accuracy in difficulty-level generation.

3.4 Preliminary Trial and Lessons Learned

Our generated questions have focused on program tracing in a multiple-choice format. Both the correct answer and distractor answers are determined by virtually executing the generated code in the way explained in Section 3.2. As mentioned earlier, generated questions are delivered to students on an app downloaded to their smartphones. This app is part of the mobile platform developed by this research group for delivering practice questions for any course (a similar experiment with Biology questions is underway). The platform tracks the performance of every individual on every question, and recycles questions (after a gap in time) that a learner got wrong, while also providing an instructor with a dashboard of student activity.

Our pilot was performed at the end of the Fall 2017 and Spring 2018 semesters at our university. The course in both cases was the

standard CS-1 course (in Java) offered to majors and non-majors alike. The questions were generated and given to student volunteers in the course ($N = 79$) about two weeks before the final exam, as practice for the final exam. Exercises focused on *for-loops*, *conditionals*, *for-loops* with *conditionals*, and *for-loops* with *for-arrays*. The initial question generation process resulted in a set of 200 questions. Some of the original questions (complex conditionals) were discarded due to their length, which did not fit the course objectives. In future iterations, maximum problem length may be added as an input parameter.

3.5 Types of generated questions

3.5.1 Conditionals . The evaluation of expressions, as used in conditionals, to their final Boolean values prepares students to learn more complex programming blocks such as *for-loops*. In these challenges, students must correctly follow the path of a variable through a series of manipulations. While nesting conditional statements is one way to increase the difficulty level, an excessive use of nesting results in spaghetti-looking code. Therefore, the generation also combines conditionals with loops.

3.5.2 For-Loops . Understanding simple and nested for-loops, especially those that manipulate variables across scope, encourages students to build upon their knowledge of execution order and conditionals. Eventually, the hope is that students come closer to viewing simple *for-loops* as an operation, instead of having to unroll each loop during execution.

3.5.3 For-Loops and Conditionals . Combining a *for-loop* with *conditionals* is a staple of programming, and therefore useful as a practice exercise. The conditionals can exist both outside and inside the loop, and can be nested. The resulting problems usually have distractors that students find challenging, especially if the iteration variable is used in generating the distractor.

3.5.4 For-Loops and For-Arrays . Since arrays are important and are often the first type of data structure seen by students, GenCODE generates questions that combine loops and arrays. These questions are particularly useful when the array itself is modified inside the loop, and when the loop iteration variable is featured in the array references.

4 RESULTS

In the following parts, we discuss the results of the approach used to set difficulty, and the responses from the students that participated.

4.1 Difficulty and Executable Blocks

We use the term *executable-block* to approximate what is sometimes called a *basic block* in compiler terminology: a piece of code that executes linearly without branching. Thus, a compiled program is a collection of basic blocks connected through jump instructions. Since we are using a tree representation for a generated program, we use the term executable-block because it also captures smaller elements such as declaration and expressions. For example, a *for-loop* has at least four executable blocks that make up its declaration, comparison, and stepping logic. Figure 2 shows the relation between the target difficulty (entered by the instructor) difficulty and the total number of executable blocks, organized by topic (In the Figure, *ARRAY* represents the *for-array* topic).

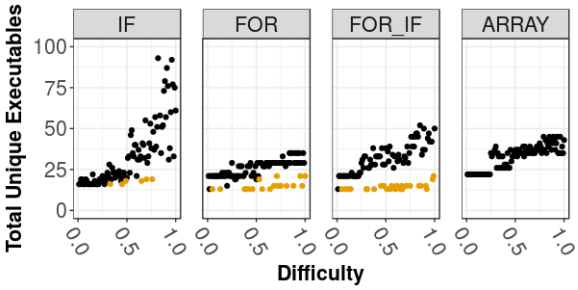


Figure 2: Difficulty Input Vs Number of Executables for each topic. Light points can be identified and discarded for being too easy.

As one would expect, the size and complexity of the programs increase as the input difficulty parameter increases. This, in addition to the patterns, allows the system to generate sets of problems that vary in difficulty and topic and that can be easily adjusted to fit a desired challenge level. Given the probabilistic nature of the question generation, some questions do not include programming blocks that provide added complexity and therefore remain "easy". These can be seen as the sets of points with few executable blocks, despite the increasing target difficulty. These cases are easily identifiable and future versions of GenCODE will adjust for this issue by eliminating them.

4.2 Number of Useful questions

Even without discarding the trivial cases shown as lighter points in Figure 2, the ratio of useful questions was high. The ratios for each question type are shown in Table 1. This means that for the *conditionals* and for the *for-loops*, the instructor will have to discard half of the generated questions, while for the *for-loops* with *conditionals*, and for *for-arrays*, the instructor must discard about one in twenty.

Table 1: Ratio of Useful questions

IF	53.6%
FOR	54.1%
FOR_IF	96.6%
FOR_ARRAY	93.3%

4.3 Generated vs. Perceived Difficulty

In order to determine the degree to which input (or generated) difficulty adhered to perceived difficulty, we conducted a small survey where 12 experienced programmers where asked to label 36 questions as being "easy", "medium", or "hard". We compared the actual difficulty and the assigned label. Two instances were left unlabeled for a total of 430 labeled difficulties. As can be seen in Table 2 and Figure 3, the distribution of difficulty labels is reasonably aligned with the desired difficulties in the sense that the peak of each of the label categories lies on the actual difficulty.

Table 2: Difficulty Assignment Table

		Labeled		
		Easy	Medium	Hard
Created	Easy	112	16	2
	Medium	66	75	15
	Hard	31	70	43

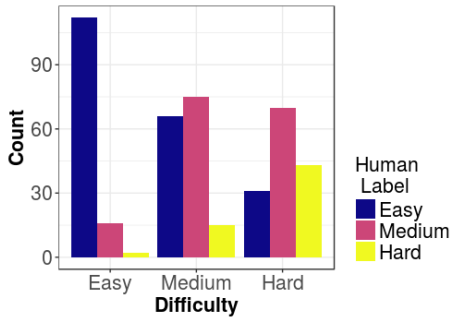
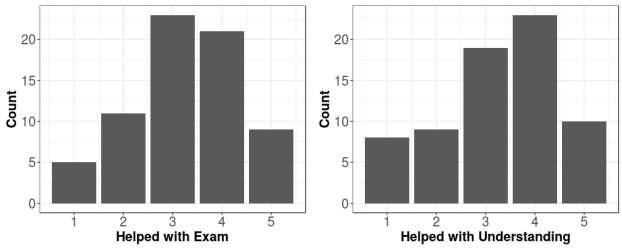


Figure 3: Difficulty VS Human Labeling

4.4 Student Response

A survey of student experience was collected at the end of the semester. Figure 4 shows the distribution of responses when students where asked "To what extent did the practice exercises help you prepare for the exam?" (in Figure 4a); and when they were asked "To what extent did the practice exercises help uncover some gap in your understanding (that perhaps led you to review some concept)?" (in Figure 4b).



(a) (b)

Figure 4: Perceived effect of using the system on the final exam 4a and for conceptual understanding 4b.

As can be seen in Figure 4, while there was a generally positive effect of the use of GenCODE to help prepare students for the final exam, the main effect was on helping students identify erroneous concepts. This suggests the possibility of implementing such as system from the start of a semester, rather as a simple practice and review tool. These conclusions are supported by the responses from students where 75% preferred to have the tool introduced early in the course; 3% preferred somewhere in the middle; and 22% prefer it as a practice tool for the final exam.

5 DISCUSSION

The stochastic tree-based generation algorithm is a compromise between using a database of templates and pure generation from the target language's grammar. The latter produces too many unrealistic and unwieldy code snippets, while the former incurs instructor overhead. The stochasticity in generation enables scaling and repeated practice of the same level of difficulty.

The higher-level tree-based approach allows topic combinations as well as a reasonable way to achieve target difficulty levels, a necessary condition for reliably generating questions in a gradation of difficulty. In terms of difficulty variation, GenCODE uses the pattern tiers and probabilistic nesting and concatenation of programming blocks. This procedural generation system resulted in a reasonably close correlation between the generated and human-labeled difficulties. Combining the analysis of the results obtained with the adjustable method of increasing difficulty can allow the instructor to develop questions in the student's zone of proximal development.

Another benefit of the stochastic tree approach is the fact that it can be easily extended to other languages. The blocks represent programming concepts that are combined to produce the tree, which is used, in turn, to obtain the correct answer and the appropriate distractors. The only element that needs to be adjusted is the module that generates the sample code to produce the actual question, and small changes to the execution engine.

Our analysis of the trial showed that students get questions wrong on all difficulty tiers with similar frequencies, which points to the effectiveness of the distractor generation.

An additional aspect of interest is the possibility of using the tree of a programming question to infer difficulty. While the initial results linking the number of executable blocks and input difficulty point towards a correlation between the two, there still remains the question of whether or not the reverse is true: if the parsing of an existing problem into its constituent unique executable blocks lend itself to an extraction of difficulty.

While the original intention was to use the system as a practice tool for the final exam, a significant amount of students found it extremely useful in helping them detect, and then correct, conceptual errors. Most of the user comments focused on the utility of "walking through" problems, especially for the topic of *loops* and *nested loops*. Other sample comments:

- "The whole structure of the problems was useful to be able to mentally walk through a program instead of just clicking 'run' "
- "Help finding the output of a complex method"
- "I discovered how much improvement I needed in interpreting other's code"
- "When helping me to find where I need to put the most studying in"
- "It helped me practice reading code instead of just writing it"

In the course of the study, we discovered that the produced output still needs some work to be indistinguishable from handcrafted problems. Details in the produced questions seem to suggest to the user that a question has been artificially generated. One example given was: "A few questions had expressions like "if a >

a", which seemed like something a human would be unlikely to include". While the tool was found to be beneficial for practice and for concept clarification, by presenting more realistic instances of programming questions, the usefulness of the tool would increase.

Note that this project has not conducted a learning effectiveness study, for which we rely on the works of [6, 23]. Nonetheless, further study is warranted to compare the effectiveness of particular types of questions.

6 CONCLUSIONS AND FUTURE WORK

This paper explored the potential of automated generation of multiple-choice program-tracing questions using a stochastic tree-based approach. The approach offers the advantage of high scalability, correlation with desired difficulty levels, and easy portability to other languages. A preliminary trial resulted in supporting these goals, along with positive feedback from students.

In the future, we intend to quantitatively evaluate the learning effectiveness of the tool when used as support for conceptual clarification when introduced at the beginning of a course and not just as an exam practice resource. In addition, we plan on refining the procedural question generation to detect and discard those questions that present syntax that is clearly artificial.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the efforts of Michael Goddard, a GW undergraduate who developed an exploratory prototype for his senior project, and Jennifer Hill, who developed much of the server/app combination that was used to distribute the generated questions. Part of this effort was supported through NSF award 1347516.

REFERENCES

[1] Stina Bridgeman, Michael T. Goodrich, Stephen G. Kobourov, and Roberto Tamassia. 2000. PILOT: an interactive tool for learning and grading. ACM Press, 139–143. https://doi.org/10.1145/330908.331843

[2] Peter C Brown, Henry L Roediger III, and Mark A McDaniel. 2014. *Make it stick*. Harvard University Press.

[3] Peter Brusilovsky and Sergey Sosnovsky. 2005. Individualized exercises for self-assessment of programming knowledge: An evaluation of QuizPACK. *Journal on Educational Resources in Computing* 5, 3 (Sept. 2005), 6–es. https://doi.org/10.1145/1163405.1163411

[4] G. Dancik and A. Kumar. 2003. A tutor for counter-controlled loop concepts and its evaluation, Vol. 1. IEEE, T3C_7–T3C_12. https://doi.org/10.1109/FIE.2003.1263331

[5] Debzani Deb, Mohammad Muztaba Fuad, and Mallek Kanan. 2017. Creating engaging exercises with mobile response system (MRS). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 147–152.

[6] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*. ACM, 20–29.

[7] K Anders Ericsson, Robert R Hoffman, Aaron Kozbelt, and A Mark Williams. 2018. *The Cambridge handbook of expertise and expert performance*. Cambridge University Press.

[8] Eric Fernandes and Amruth N. Kumar. 2004. A tutor on scope for the programming languages course. *ACM SIGCSE Bulletin* 36, 1 (March 2004), 90. https://doi.org/10.1145/1028174.971332

[9] Michail N Giannakos. 2013. Enjoy and learn with educational games: Examining factors affecting learning performance. *Computers & Education* 68 (2013), 429–439.

[10] I-Han Hsiao, Peter Brusilovsky, and Sergey Sosnovsky. 2008. Web-based parameterized questions for object-oriented programming. Association for the Advancement of Computing in Education (AACE), 3728–3735.

[11] I.-H. Hsiao, S. Sosnovsky, and P. Brusilovsky. 2010. Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming: Adaptive navigation support in E-Learning. *Journal of Computer Assisted Learning* 26, 4 (July 2010), 270–283. https://doi.org/10.1111/j.1365-2729.2010.00365.x

[12] Petri Ihantola, Juha Helminen, and Ville Karavirta. 2013. How to study programming on mobile touch devices: interactive Python code exercises. ACM Press, 51–58. https://doi.org/10.1145/2526968.2526974

[13] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A mobile learning application for parsons problems with automatic feedback. ACM Press, 11–18. https://doi.org/10.1145/2401796.2401798

[14] Amruth N. Kumar. 2005. Generation of problems, answers, grade, and feedback—case study of a fully automated tutor. *Journal on Educational Resources in Computing* 5, 3 (Sept. 2005), 3–es. https://doi.org/10.1145/1163405.1163408

[15] Amruth N. Kumar. 2005. Results from the evaluation of the effectiveness of an online tutor on expression evaluation. *ACM SIGCSE Bulletin* 37, 1 (Feb. 2005), 216. https://doi.org/10.1145/1047124.1047422

[16] Amruth N Kumar. 2015. Solving code-tracing problems and its effect on code-writing skills pertaining to program semantics. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 314–319.

[17] Amruth N. Kumar. 2018. Epplets: A Tool for Solving Parsons Puzzles. ACM Press, 527–532. https://doi.org/10.1145/3159450.3159576

[18] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. ACM Press, 161. https://doi.org/10.1145/1562877.1562930

[19] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. ACM Press, 101–112. https://doi.org/10.1145/1404520.1404531

[20] Solomon Sunday Oyelere, Jarkko Suhonen, and Teemu H. Laine. 2017. Integrating parson's programming puzzles into a game-based mobile learning application. ACM Press, 158–162. https://doi.org/10.1145/3141880.3141882

[21] Dale Parsons and Patricia Haden. [n. d.]. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses, Vol. Vol. 52. Australian Computer Society, Inc., Australia, 157–163.

[22] Ferran Prados, Imma Boada, Josep Soler, and Jordi Poch. 2005. Automatic generation and correction of technical exercises. In *International conference on engineering and computer education: Icece*, Vol. 5.

[23] Laura Zavala and Benito Mendoza. 2018. On the Use of Semantic-Based AIG to Automatically Generate Programming Exercises. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 14–19.