Current State and Next Steps on Automated Hints for Students Learning to Code

Daniel Toll
Department of Computer Science, Linnaeus University,
Kalmar, Sweden
daniel.toll@lnu.se
ORCID-id: 0000-0001-5335-5196

Anna Wingkvist
Department of Computer Science, Linnaeus University,
Växjö, Sweden
anna.wingkvist@lnu.se
ORCID-id: 0000-0002-0835-823X

Morgan Ericsson
Department of Computer Science, Linnaeus University,
Växjö, Sweden
morgan.ericsson@lnu.se
ORCID-id: 0000-0003-1173-5187

***Abstract*—The core of this work-in-progress is that the best way to learn how to code is to practice by solving problems. However, if students have trouble with this, they can get frustrated and give up. Automated Tutoring Systems (ATS) aim to provide hints to help them solve the problems they encounter. Many of the existing systems offer general hints, e.g., "check the conditional statement" or help the student interpret the compiler or test-case errors. While this can be useful, we think that an ATS should provide interactive and specialized feedback for each program. We snowballed through publications on promising ATS and found that there are several such systems (in 27 publications), but we could also identify many challenges and that our requirements were not met by any existing system. For example, few of them work on general-purpose programming languages, e.g., Java, or scale to realistic problems consisting of multiple methods and classes. From the search, we find ATS based on Automated Program Repair (APR) shows the most promise. However, while program repair has the potential to generate specialized hints to help guide the student to a working state, studies that looked into these have identified further challenges. For example, many APR ATS tools only show the repaired program to the students, who then have to compare and modify their program accordingly. Another issue is that APR generally only modifies a few lines, so if the student solution is far from correct, the repair might fail. This can be solved by partial repair, i.e., the program is repaired so at least one additional test-case passes. While this increases the repair rate, it might make hints more difficult or point the students in a non-obvious or even "wrong" direction. The APR can take several minutes, which also makes it unsuitable for interactive ATS. We take a design science approach to define an ATS based on APR that attempts to address the identified challenges. We give a review of the state-of-the-art for the required components, e.g., APR, how to generate hints from differences between two programs. From this, we suggest a three-step roadmap; 1. identify suitable APR-tools, 2. construct an oversized test-suite, and 3. adopt APR to the tutoring context.**

***Keywords*—programming, program repair, tutor tools**

I. Introduction

We noticed that many of our students do not have enough problem-solving and programming skills, even after completing the introductory programming courses with passing grades. We think that the best way to learn how to program is by practicing solving problems, and this is where our introductory courses fail; all practice problems provided are graded, so we are, in essence, limited by what we can manually grade and provide feedback on. The number of students has increased, and more courses are either moving to hybrid-mode or entirely online, which further stretches our resources.

We want our students to solve significantly more programming problems than they do today, to increase their confidence and provide the necessary foundation for more advanced courses where the problems are more complex, resulting in larger codebases, and require additional Application Programming Interfaces (API), e.g., sockets or threading. Multiple tools provide automatic grading, but if students get stuck and cannot progress, they can get both frustrated and bored [1]. We have limited options to help our students when this happens. Tutoring is often restricted to specific days and hours, so it can take some time before the students get help. Peer instruction can reduce the wait, but peers might provide the wrong or too much help, increasing the frustration or limiting the benefit. Similarly, it can be challenging to find help online, e.g., Q&A sites such as Stack Overflow, since novice programmers often need contextualized answers, and general solutions can make the situation worse. In 2018, from a literature review [2], it was reported that the two most common motivators for new tools to be introduced in programming courses are handling large student groups and students' requests for timely feedback. These two motivators are in the foreground for us as well.

In this work-in-progress, we report our search for tools that give students automated hints when learning to code. From the review of tools, we identified several challenges. To counter these, we take a design science approach [3] to define an Automated Tutoring System (ATS) based on Automatic Program Repair (APR). We provide a roadmap to advance the development of an APR-based ATS.

II. Challenges in Automated Tutoring

An ATS aims to provide hints to students to help them solve the difficulties they encounter. While searching for suitable tools that can be used for our programming courses to provide timely hints, we identified several shortcomings. In a systematic literature review covering 101 ATS tools, Keuning, Jeuring, and Heeren [4] found that only a few of the tools provided "good" hints on how to proceed for students that are stuck. Many of the existing systems use general hints, e.g., "check the conditional statement" or help the student interpret

the compiler or test-case errors. While this can be helpful, we think that an ATS should provide interactive and specialized feedback for each program; only providing a test report from an automated test-suite is probably not enough when a student does not know how to proceed.

> **Challenge:** Many of the existing ATS provide general hints that might not help a student stuck on a specific problem.

Instead, we prefer an ATS to give meaningful hints on how to proceed from the current code state into a new state closer to a correct goal state. Such hints are named "knowledge on how to proceed" [4], "goal-directed", or "next step hints" [5]–[7]. These hints "unlocks" a part of the solution just like a worked-out solution by giving the student a hint that will take them closer to solving the problem. These may also lead the student away from things that will not work.

To find studies describing suitable tools, we used the snowballing technique. This approach was inspired by Wohlin [8] and is done to find studies by following citations from an initial set of studies. For snowballing to work, the initial set must include at least one study from each of the different "islands" of authors that do not cite each other. To get an as wide initial set as possible, we used three databases: Google Scholar, ACM Digital Library, and Scopus. The search strings were based on terms found in review articles, including programming, tutor, tool, environment, hint generation, generating hints, hint, hints, hinting, adaptive feedback, and personalized feedback. The search queries were customized to each search engine depending on search capabilities to get a high rate of included studies while also reducing the number of false positives. To evaluate, we used a set of eight previously known studies (Gold Standard) that had to be in the combined search result from the three databases. Note that we did not base the search terms on the content of these studies but the review articles.

Since it is common to get an unmanageable amount of results on Google Scholar, we used a restrictive but wide query that only included papers with the terms hint, hints, or feedback combined with programming in the title. For ACM Digital Library, we used a similar query but adopted it for that database. The Scopus search engine allows complex queries, and we used the following: *(ALL (programming) AND TITLE-ABS-KEY ((tutor OR tool OR environment)) AND TITLE-ABS-KEY ("hint generation" OR "generating hints" OR "hinting" OR "adaptive feedback" OR "personalized feedback"))*.

The combined search resulted in 336 studies with a low amount of duplicates (39). All studies in the Gold Standard were found in this search result. To select our initial search set, we only included studies that describe automated programming tutor tools for general-purpose programming languages that do not restrict the programmer to specific solutions and offer goal-directed hints or feedback. In the initial search result, we found 15 studies. This increased to a final count of 27 studies over three iterations of snowballing. The snowballing stopped when no new studies were found.

The 27 publications from 2009–2019 describe nine different tools that generate "next step"-hints. Some tools use a set of authored model-goal-solutions to automatically generate a set of valid step-by-step refinements (also called strategies) [9]–[16]. When a student asks for a hint or deviates from the path, a hint can be shown to direct the student back on track. In some of the ATS-tools, the model solutions can be annotated to allow variation or to write messages displayed to the students as hints [15]. This gives the tutor control over what the correct solution is, but requires them to have vast experience and suffers from scalability issue when the programming assignments grow complex.

> **Challenge:** Authored hints quickly become unmanageable when the problems become more complex. Such hints require much effort to create, and the tutor needs to predict what errors students are likely to make, and that requires experience.

Data-driven ATS-tools that give "next step"-hints rely on submissions by other students. These systems can guide a student to any solution previously submitted for the task [5], [7], [17]–[22]. The closest solution is found by a comparison function that calculates the distance between the submitted code and all solutions seen before. Since the set of possible solutions is large, normalization methods are used to match both submitted code and the solutions so that semantically identical code can be matched. Natural variations, such as identifier names and white-spaces, can thus be ignored [5], [7], [15], [23]. When a target solution is identified, a path must be constructed to decide which actions the student has to do to reach that goal state and in what order those actions should be presented as hints. Previous work used a Markow Decision Process [24] to decide the next state, or by ordering the submitted program states into paths in a solution space using distance metrics between code states (e.g., Levenshtein distance, Tree Edit distance).

> **Challenge:** Data-driven hint generation relies on student submissions, making it difficult to control the quality of the hints. The approach also does not scale well, since semantic variations will make it very hard to match one submission with any solution seen before, no matter the amount of normalization.

A potential problem with data-driven hinters is that a solution submitted by a peer-student might not be the best solution or something we as tutors would recommend learning from [4]. We have seen our students produce sub-optimal solutions that, by pure luck, pass the test-suite rather than out of skill (sometimes due to the lack of tests). Another problem is that the solution space is vast, and previously submitted solutions might not be enough to capture all possible variations, especially in the beginning, when no solutions have been collected (i.e., the cold start problem).

Le and Pinkwart [25] classified programming exercises into different degrees of openness for the students to come up

with their solutions. They define three classes with increasing openness in the solution space. Class one only supports a single solution, while class two supports some variation in, for example, variable names, or loop types, but only a single solution-strategy (algorithm) is supported. The third class supports several solution strategies for the same problem.

Hinting systems generally restrict the solutions created by the students to those provided by the tutor even if some variation is allowed [9]–[16]. We argue that tools supporting more variation is a must when working with larger programming problems such that are introduced in university courses beyond introductory programming. A successful ATS must, therefore, support open-ended programming tasks. However, most tools we have seen so far that can automatically generate hints seem to be evaluated on tiny problems, i.e., only a few lines of code [6], [10], [21], [22], [26], [27]. Also, it is quite common for the hinting tools not to support the full programming language [15] or lack support for classes [26].

Challenge: Existing automated tutoring systems restrict the solution space too much, especially when programs grow more complex. Many systems only support subsets of programming languages, e.g., Java without classes.

Finally, we note that many of the tools are not publicly available or are still prototypes. Few of the existing ATS tools work on general-purpose programming languages, e.g., Java. Ihantola, Ahoniemi, Karavirta, *et al.* [28] found it rare to see tools used outside of their home institution and theorize that the reason is that they are either part of a Ph.D. work or created to solve a particular problem in a particular setting.

Challenge: Most automated tutoring systems are research prototypes that are not suitable for general use.

III. Automated Program Repair

The main problem with data-driven hint generation is that it does not scale when the solution space grows. APR uses failed tests to identify faults. It then tries to generate patches for the faults automatically so that all tests in the suite succeed. Generally, an APR-tool works in three steps: First, in the **fault localization** step, it identifies suspicious locations in the code that are likely to contain a fault given the failing test. Second, in the **patch generation** step, the source code in the location is modified into a patch candidate, for example, by applying a set of mutations to the suspicious code such as removal of statements, modification of if-conditions, or inserting new statements. Some tools search for common patches in databases, or similar code fragments in source code repositories, and use that code to generate patches [29]. Third, in the **patch validation** step, when a patch candidate is found, it must be validated against the test-suite. This is where the patched source code is tested against the test-suite.

There exist quite a few automatic repair tools, and 43 are listed on the Program-repair.org website[1]. There are different APR-tools for languages such as C/C++, Eiffel, Java, and Python. These tools range over many different strategies for automating patch generation; for overview see Liu, Wang, Koyuncu, *et al.* [29], which divides the tools into heuristic-based repair, constraint-based, and template-based approaches. While APR-tools are still cannot find all bugs, the number of bugs they find increases over time [29].

APR has the potential to be used in education by generating specialized hints to help guide the student to a working state. However, studies that investigate these have identified some shortcomings as well [30], [31]. APR-tools are designed to fix minor problems in large code-bases; thus, their output is a patched source code. To generate and validate patches takes time and computational resources. For example, when Yi, Ahmed, Karkare, *et al.* [31] used APR tools in an ATS, they had to limit their experiment's search time to 15 minutes. Even if a successful search averaged around one minute, only 31 % of the programs were repaired within the time-limit.

Challenge: APR can take several minutes, which also makes it unsuitable for interactive ATS.

When Yi, Ahmed, Karkare, *et al.* [31] showed APR-generated patches to novice students, they were unable to make efficient use of the suggested patches. In the tutoring context, we would like the tool to output helpful feedback. The means of presenting these hints in a way that is beneficial for learning is important. For example, instead of revealing the solution in one step, we would like it to give hints that allow the student to proceed gradually, for example, with next-step hints.

Challenge: Many APR ATS tools show the repaired program to the students, who then have to compare the two and modify their program accordingly.

Another issue is that APR generally only modifies a few lines, so if the student solution is far from a correct solution, the repair might fail. Partial repair [31] can solve this, i.e., the program is only repaired to the degree that at least one additional test-case passes. While this increases the repair rate, it might make the hints more difficult to understand or direct the students in a non-obvious and even "wrong" direction.

Challenge: ATS APR does not work well if the student solution is far from a correct solution. Partial repair helps but can result in non-obvious and even wrong hints.

IV. A Roadmap

We want our students to practice designing object-oriented programs with multiple methods and classes, but these seem to be entirely out of scope for existing ATS. Data-driven approaches show promise, but it is clear that they will not scale to our needs. The problems we are interested in allow several solution strategies for the same problem, which would make the solution space large. Even with access to several solutions, the distance between will likely be too wide. In our opinion, APR has a lot in common with the data-driven

[1]https://program-repair.org (accessed on 2020-04-09).

hinting approaches, but have several advantages. They focus on real-world scale programs and programming languages, and the research community is more active. However, APR-tools are not designed for tutoring, and for example Yi, Ahmed, Karkare, *et al.* [31], have exposed several problems. Our roadmap has the end-goal of developing an open-source ATS tool based on APR capable of producing towards-solution-hints. To achieve that goal, we define sub-goals that are based on the challenges identified. Each step is iterative, addresses one or more problems, and will be evaluated.

1) Evaluate cutting edge APR-tools: We aim to start by comparing the most promising existing APR-tools available to get an overview of their capabilities in the tutoring domain. We will start by investigating the publications on cutting-edge tools, compare their capabilities, availability, repair rate, language support, and suitability for adoption to the tutoring domain. We will start from the list of automatic programming tools and benchmarks from program-repair.org. To get an in-depth understanding of the most promising tools, we need to try them on realistic data. We expect to have to establish a benchmark of student solutions. We aim to create this from existing datasets to make sure that the solutions are realistic.

2) Over-sizing the test-suite: We intend to investigate the repair rate on realistic student solutions when we have found one or more promising APR-tools. We expect that incorrect patches will be the main obstacle for APR-tools to be used to generate hints to students since these risk to confuse students further. This was noted by Yi, Ahmed, Karkare, *et al.* [31]. Partial repairs, patches that fix one more test while maintaining the rest, helped make APR suitable for ATS, but this needs to be a carefully guided process, to avoid generating "wrong" hints. We intend to investigate how to create groups of tests for the same feature that are always run together to avoid partial repairs that are misleading. We plan to address this challenge by over-sizing the test-suite so that more test-cases than usual cover every part of the tested program. To do this, we plan to use techniques from Software Testing, i.e., a model solution to derive expected values for new test-cases, and Random Testing (or Model-Driven Testing) to generate input, thus enabling the generation of many more test-cases.

To measure the test-suites effectiveness and be able to prune redundant tests to speed up the testing, we must carefully evaluate the fault detection capability. Standard coverage methods such as basic coverage is a starting point but might be too closely connected to the model solution and might not be a good measure of how well a test suite works for an unknown source code such as those submitted by our students. Mutation testing has matured over the last years, and thus we may also benefit from Mutation Score [32] or Checked Coverage [33], where only instructions that affected the value of the assert statement is counted as covered. When we have an evaluation measure for the test-suites, we intend to start testing this on the available student submissions from [31].

3) Adopt APR to the tutoring context: We intend to adopt the most promising APR-tool (decided in the testing stage) to the tutoring context. We expect this to contain many challenges on its own, and thus we only outline the most critical steps. We need to turn patches into step-by-step changes that can be offered as hints to the students, allowing progression in steps from their submitted code into a valid solution. These step-by-step changes need to be translated into a natural language description that the student can understand. We expect that steps might need to be combined or split to create descriptions that the student can follow, e.g., an off-by-one-error might require several changes but can be explained in a single hint.

Another issue is the vast search space that the APR-tools need to deal with. We expect that our problem makes this search harder, not easier, and therefore anticipate that the compute time for a hint can be too long for interactive use. We plan to investigate this from both a computational and an interaction perspective. Possibly we can reduce the time required to compute the hint, e.g., by using iterative searches and computations. While many student solutions will differ, we reckon some overlap, which might enable us to reuse results. We will also explore ways to accelerate the processing time, e.g., via parallel implementation. However, we still expect compute time to be significant, so we plan to investigate how "instant" a hint needs to be and how long delays students are willing to accept.

V. Summary

In our search for an ATS, we were unable to find one suitable for our programming courses. We describe the challenges that must be faced to implement a tool based on APR, including scalability issues, adoption of APR-tools to the tutoring context, and faulty patches. We suggest a three-step roadmap to attack these challenges; 1. identify suitable APR tools, 2. construct an oversized test suite, and 3. adopting APR to the tutoring context.

References

[1] N. Bosch and S. D'Mello, "The affective experience of novice computer programmers," *Int. J. of Artif. Intell. in Educ.*, vol. 27, no. 1, pp. 181–206, Mar. 2017.

[2] A. Luxton-Reilly, Simon, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, and C. Szabo, "Introductory programming: A systematic literature review," in *Proc. Companion of 23rd Annual Conf. on Innovation and Technology in Computer Science Educ.*, ACM, 2018, pp. 55–106.

[3] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[4] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, 3:1–3:43, Sep. 2018.

[5] K. Rivers and K. R. Koedinger, "Automating hint generation with solution space path construction," in *Int. Conf. on Intell. Tutoring Systems*, Springer, 2014, pp. 329–339.

[6] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: A self-improving Python programming tutor," *Int. J. of Artif. Intell. in Educ.*, vol. 27, no. 1, pp. 37–64, 2017.

[7] K. Rivers and K. R. Koedinger, "Automatic generation of programming feedback: A data-driven approach," in *1st WS on AI-supported Educ. for Computer Science*, vol. 50, 2013.

[8] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proc. 18th Int. Conf. on Evaluation and Assessment in Software Engineering*, ACM, 2014, 38:1–38:10.

[9] J. Jeuring, A. Gerdes, and B. Heeren, "A programming tutor for haskell," in *Central European Functional Programming School*, Springer, 2011, pp. 1–45.

[10] A. Gerdes, J. Jeuring, and B. Heeren, "An interactive functional programming tutor," in *Proc. 17th Annual Conf. on Innovation and Technology in Computer Science Educ.*, ACM, 2012, pp. 250–255.

[11] A. Gerdes, B. Heeren, and J. Jeuring, "Teachers and students in charge," in *21st Century Learning for 21st Century Skills*, A. Ravenscroft, S. Lindstaedt, C. D. Kloos, and D. Hernández-Leo, Eds., Springer, 2012, pp. 383–388.

[12] J. Jeuring, A. Gerdes, and B. Heeren, "Ask-elle: A haskell tutor," in *21st Century Learning for 21st Century Skills*, A. Ravenscroft, S. Lindstaedt, C. D. Kloos, and D. Hernández-Leo, Eds., Springer, 2012, pp. 453–458.

[13] J. Jeuring, L. T. van Binsbergen, A. Gerdes, and B. Heeren, "Model solutions and properties for diagnosing student programs in ask-elle," in *Proc. Computer Science Educ. Research Conf.*, ACM, 2014, pp. 31–40.

[14] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: An adaptable programming tutor for haskell giving automated feedback," *Int. J. of Artif. Intell. in Educ.*, vol. 27, no. 1, pp. 65–100, 2017.

[15] H. Keuning, B. Heeren, and J. Jeuring, "Strategy-based feedback in a programming tutor," in *Proc. Computer Science Educ. Research Conf.*, ACM, 2014, pp. 43–54.

[16] A. Gerdes, B. Heeren, and J. Jeuring, "Constructing strategies for programming.," in *CSEDU (1)*, 2009, pp. 65–72.

[17] T. Lazar, A. Sadikov, and I. Bratko, "Rewrite rules for debugging student programs in programming tutors," *IEEE Transactions on Learning Technologies*, vol. 11, no. 4, pp. 429–440, 2017.

[18] T. Lazar and I. Bratko, "Data-driven program synthesis for hint generation in programming tutors," in *Int. Conf. on Intell. Tutoring Systems*, Springer, 2014, pp. 306–311.

[19] S. Chow, K. Yacef, I. Koprinska, and J. Curran, "Automated data-driven hints for computer programming students," in *25th Conf. on User Modeling, Adaptation and Personalization*, ACM, 2017, pp. 5–10.

[20] K. Wang, B. Lin, B. Rettig, P. Pardi, and R. Singh, "Data-driven feedback generator for online programing courses," in *Proc. 4th Conf. on Learning@ Scale*, ACM, 2017, pp. 257–260.

[21] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *SIGPLAN Notices*, ACM, vol. 53, 2018, pp. 481–495.

[22] T. W. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes, "The impact of data quantity and source on the quality of data-driven hints for programming," in *Int. Conf. on Artif. Intell. in Educ.*, Springer, 2018, pp. 476–490.

[23] P. Xu and Y. Chee, "Transformation-based diagnosis of student programs for programming tutoring systems," *IEEE TSE*, vol. 29, pp. 360–384, May 2003.

[24] T. Barnes and J. Stamper, "Toward automatic hint generation for logic proof tutoring using historical student data," in *Intell. Tutoring Systems*, B. P. Woolf, E. Aïmeur, R. Nkambou, and S. Lajoie, Eds., Springer, 2008, pp. 373–382.

[25] N.-T. Le and N. Pinkwart, "Towards a classification for programming exercises," in *2nd WS on AI-supported Educ. for Computer Science*, Jan. 2014.

[26] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *Sigplan Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[27] S. Gulwani, "Example-based learning in computer-aided stem education," *Commun. ACM*, vol. 57, no. 8, pp. 70–80, 2014.

[28] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proc. 10th Koli Calling Int. Conf. on Computing Educ. Research*, ACM, 2010, pp. 86–93.

[29] K. Liu, S. Wang, A. Koyuncu, K. Kim, P. Wu, J. Klein, X. Mao, Y. Le Traon, T. Bis-Syandé, and D. Kim, "On the efficiency of test suite based program repair : A systematic assessment of 16 automated repair systems for java programs," Feb. 2020, [Online]. Available: http://hdl.handle.net/10993/42854.

[30] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019.

[31] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proc. 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 740–751.

[32] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[33] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *4th IEEE Int. Conf. on Software Testing, Verification and Validation*, 2011, pp. 90–99.