An Epistemic Model-Based Tutor for Imperative Programming

Amruth N. Kumar [0000-1111-2222-3333]

Ramapo College of New Jersey, Mahwah NJ 07430, USA
amruth@ramapo.edu

Abstract. We developed a tutor for imperative programming in C++. It covers algorithm formulation, program design and coding – all three stages involved in writing a program to solve a problem. The design of the tutor is epistemic, i.e., true to real-life programming practice. The student works through all the three stages of programming in interleaved fashion, and within the context of a single code canvas. The student has the sole agency to compose the program and write the code. The tutor uses goals and plans as prompts to scaffold the student through the programming process designed by an expert. It provides drill-down immediate feedback at the abstract, concrete and bottom-out levels at each step. So, by the end of the session, the student is guaranteed to write the complete and correct program for a given problem. We used model-based architecture to implement the tutor because of the ease with which it facilitates adding problems to the tutor. In a preliminary study, we found that practicing with the tutor helped students solve problems with fewer erroneous actions and less time.

Keywords: Programming Tutor, Imperative Programming, Model-Based Architecture.

Introduction. Numerous tutors have been developed to help students learn to write code in high-level languages [15,16,17] such as LISP [5], Haskell [27] and Prolog [21], imperative languages such as Pascal [6], Java [24, 25, 26] and C# [23], and scripting languages such as Python [22, 28]. We built a tutor for imperative programming in the popular language C++ for use by introductory programming students.

When learning to program, students need explicit instruction on formulating the algorithm [14]. Several flowchart-based programming environments have been developed to help improve the algorithm formulation skills of students (e.g., [1, 2, 13, 19, 20, 29]). ProPL [8] uses natural language to help students write pseudocode.

Several systems have been reported that integrate all three stages of programming, viz., algorithm formulation, program design and coding, including LISP Tutor for LISP [5] and PROUST [6], BRIDGE [7] and GPCEditor [9] for Pascal and Guided-Planning and Assisted-Coding tutor [10,11] and J-Latte [25] for Java. Typically, these tutors deal with algorithm formulation in terms of goals and plans - the student identifies goals (what should be done next) and plans (how it should be done), before writing code for the plans.

LISP Tutor for LISP [5] and PROUST for Pascal [6] use goals and plans to diagnose the code written by the student. Instead, we use goals and plans as prompts to scaffold the problem-solving process of the student. BRIDGE [7] uses a visual intermediate language to represent the algorithm, whereas J-Latte [25] uses a visual representation. Instead, we use pseudocode as comments, which naturally belong in a program. Guided-planning and assisted coding tutor [11] provides feedback on demand during coding. It places each line of code in the program instead of asking the student to do so. In contrast, we provide immediate feedback, which has been shown to be more efficient for programming instruction [12]. In addition, we make placing each new statement in its correct location in the program the responsibility of the student. Goal-Plan-Code Editor (GPCEditor) [9] translates the plans of students into code. Instead, we have the student write the code.

Epistemic Design. Our design of the tutor is epistemic, i.e., true to real-life problem-solving and programing practice because of the following design choices:

Actions. The tutor facilitates three operations: selecting, locating and coding. In selecting operation, the student selects the appropriate step in the algorithm (e.g., which input to process next) that is translated into pseudocode in the program. The student also uses selecting operation for program design, e.g., to identify the type of control construct to use for a step in the algorithm. The student uses locating operation to compose the program, i.e., the location in the program where the next step in the algorithm should be coded. Thereafter, the student proceeds to write code for the step. The tutor does not use affordances such as drag-and-drop tiles (e.g., [25]) or flowcharts (e.g., [1, 2, 13, 19, 20, 29]) or intermediate languages [7] to design the algorithm – affordances not found in real-life programming environments.

Agency. The student is responsible for identifying the location of each step in the algorithm and program – it is not automatically determined by the tutor for the student (e.g., [11]). When coding, the student is expected to enter the frame [18] of each control construct by hand – it is not provided to the student (e.g., [25]).

Temporal Order. Trying to end algorithm design stage before going to coding (e.g., [7]) forces the novice programmer to either design based on assumptions about the code (e.g., assumptions about the control statement that will be used in a section) or code based on decisions taken too far in advance for the novice to properly appreciate (e.g., why statements must appear in a certain order in the program). In real life, programmers go back and forth between algorithm design and coding: each informing the other. In our tutor, the student goes through algorithm design, program design and coding steps in an interleaved fashion for each step.

Code Canvas. In the tutor, the student takes all the actions in the context of a single code canvas. The algorithm as pseudocode is embedded as comments within the program. So, the novice can conceptually connect each step in the algorithm with the corresponding statement(s) in the program.

Scaffolding. The tutor uses goals and plans as prompts to scaffold the student through algorithm formulation and program design instead of using them to diagnose the stu-

dent's program (e.g., [5, 6]). Every time the student selects an incorrect step in the algorithm or an incorrect choice for program design, the tutor provides immediate drill-down feedback that steers the student towards the correct step/choice. So, the student gets the opportunity to practice the process of problem-solving and programming as designed by an expert every time the student works with the tutor.

Reified Steps. During coding, the tutor provides feedback at the level of statements and expressions. When control statements such as `if-else` and `while` loop are involved, the tutor uses a script to step the student through the various components of the control statement, e.g., frame, initialization, condition, body and update for `while` loop. Such reification of steps not only makes diagnosing and providing feedback more tractable, it also trains the student to use a pedagogically effective algorithm to compose each control statement in terms of its components.

Non-deterministic. The tutor admits equivalent answers. For example, the student can select any equivalent data type for a variable (`short`, `int`, or `long`), can locate inputs in any order in the program and write a commutative expression in any order. This design acknowledges the fact that a program can be written in a multitude of ways for a problem.

Model Based Architecture. A problem is represented using a problem specification and a reference solution template. A problem specification is an annotated problem statement wherein, input and output data elements are identified, and other attributes are specified such as the expected data type for the input/output data, preferred name, etc. The reference solution template is a complete solution (i.e., program) written in BNF notation, with meta-variables for variable names (e.g., <V1>), data types (e.g., <T1>), and other program elements.

The tutor model, user interface and domain model of the tutor are all problem-independent. The tutor model includes scripts for the steps in the problem-solving process. For example, the script for input data object is: 1) Locate where the data will be input; 2) Declare the variable and 3) Input the variable. Each step in the above script may itself generate additional scripts. A declarative representation is used for these scripts so that they can be swapped to test various problem-solving processes with the same tutor. The user interface of the tutor translates each atomic step in the script into one of the following three user inputs:

1. **Select** from a drop-down menu of options. A built-in feedback server for each select action encodes drill-down feedback at abstract, concrete and bottom-out levels for each incorrect menu option selected by the student.
2. **Locate** in code by clicking in it. If the student correctly locates a statement, the tutor inserts pseudocode as a comment at that location and presents a dialog box for the student to enter the code. If the location is incorrect, the tutor provides drill-down feedback to steer the student towards the correct location.
3. Write the **code** for the next statement or expression. The domain model (described next) provides drill-down explanation for incorrect code.

The tutor determines the correctness of select actions by comparing them with the annotations in the problem statement. It determines the correctness of locate and code actions by comparing them with the reference solution provided for the problem,

The Domain Model is a model of the programming domain built using Model-Based Reasoning principles [4]: each programming construct is modeled as a component with a text representation and behavior [3]. The tutor uses the Domain Model to build a model of the student's solution, called the Program Model. The Program Model is used to generate the text representation of the student's program. It is also used to provide feedback for coding and locating actions. Each component in the Program Model is associated with a bug library relevant to that programming construct. The component generates drill-down feedback for coding errors by using the bug library to diagnose the error in the student's code. Each component is also associated with a catalog of program transformations. It uses this catalog to approve semantically equivalent code alternatives (e.g., `count++` is equivalent to `count += 1`). This ability of the Program Model to provide drill-down feedback for locating and coding actions is a significant advantage of using model-based reasoning instead of some of the other AI techniques used for modeling the domain in programming tutors such as rule-based (e.g., [5]) and constraint-based (e.g., [25]) reasoning: the drill-down feedback need not be individually specified for each problem added to the tutor. So, adding a new problem takes minimal effort – only the problem specification and reference solution template need to be specified for each new problem.

Whether it is select, locate or code action, the student cannot proceed to the next step in the algorithm until the student answers that step correctly and completely. So, by the end of the session, the student is guaranteed to have written the correct program for a given problem. For each action, the tutor provides immediate drill-down feedback at abstract, concrete and bottom-out levels, thereby ensuring that the student is never stranded at a dead-end. Given this design, the proficiency of a student is measured not in terms of the correctness of the final program, but in terms of the number of actions needed by the student to arrive at the correct program: the more actions the student takes, the less proficient the student.

The tutor is not a novel intervention for introductory programming as much as a technological facilitator of a pedagogy well understood to help introductory students learn programming – the pedagogy of practice. The more programs a student writes, the better the student becomes at the process of problem-solving and programming. The role the tutor plays is of a facilitator – it provides one-on-one scaffolding and feedback throughout the process of programming. The alternative to using the tutor in an introductory programming class would be to assign multiple programming projects on each topic, which is untenable because of the workload it entails for the instructor, not to mention the reluctance of students to engage in such labor, especially without the one-on-one feedback facilitated by the tutor. Given this, we evaluated the tutor to see whether the benefits of practice would accrue to students who use it. Preliminary results show that practicing with the tutor indeed helped students solve subsequent problems with fewer erroneous actions and in less time.

Acknowledgments. Partial support for this work was provided by the National Science Foundation under grant DUE-1432190**.**

References

1. Gomes, A., Mendes, A. J.: SICAS: Interactive system for algorithm development and simulation. In: Computers and Education - Towards an Interconnected Society, pp. 159-166 (2001)
2. Carlisle, M., Wilson, T., Humphries, J. and Hadfield, S. (2005) 'RAPTOR: a visual programming environment for teaching algorithmic problem solving, ACM SIGCSE Bulletin, CM, New York, NY, USA, Vol. 37, No. 1, pp.176–180.
3. Kumar, A.N. Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs, Intelligent Tutoring Systems (ITS 2002), Biarritz, France, June 2002, 792-801.
4. Davis, R.: Diagnostic Reasoning Based on Structure and Behavior. Artificial Intelligence, 24 (1984) 347-410.
5. Reiser, B., Anderson, J., & Farrell, R. (1985). Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pp. 8-14.
6. Johnson, W. L. (1986). Intention-Based Diagnosis of Errors in Novice Programs. Morgan Kaufman, Palo Alto, CA
7. Bonar, J. and Cunningham, R. (1988) BRIDGE: Tutoring the programming process, in Intelligent tutoring systems: Lessons learned. J. Psotka, L. Massey, S. Mutter (Eds.), Lawrence Erlbaum Associates, Hillsdale, NJ.
8. Chad Lane and Kurt VanLehn. (2005). Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education.* 15 (3). 183-201.
9. Guzdial, M., Hohmann, L., Konneman, M., Walton, C., & Soloway, E. (1998). Supporting programming and learning-to-program with an integrated CAD and scaffolding workbench. Interactive Learning Environments, 6 (1&2), 143{179.
10. Wei Jin. 2008. Pre-programming analysis tutors help students learn basic programming concepts. In Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08). Association for Computing Machinery, New York, NY, USA, 276–280.
11. Jin W., Corbett A., Lloyd W., Baumstark L., Rolka C. (2014) Evaluation of Guided-Planning and Assisted-Coding with Task Relevant Dynamic Hinting. In: Trausan-Matu S., Boyer K.E., Crosby M., Panourgia K. (eds) Intelligent Tutoring Systems. ITS 2014. Lecture Notes in Computer Science, Vol. 8474. Springer
12. Valerie Barr and Deborah Trytten. 2016. Using Turing's craft Codelab to support CS1 students as they learn to program. ACM Inroads 7, 2 (May 2016), 67–75
13. Minjie Hu, Michael Winikoff, and Stephen Cranefield. 2013. A process for novice programming using goals and plans. In Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136 (ACE '13). Australian Computer Society, Inc., AUS, 3–12.
14. Soloway, E. Learning to program = Learning to construct mechanisms and explanations. Communications of the ACM. 29 (9). Sep 1986. 850-858
15. Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. 2018. Intelligent tutoring systems for programming education: a systematic review. In Proceedings of the 20th Australasian Computing Education Conference (ACE '18). Association for Computing Machinery, New York, NY, USA, 53–62. DOI:https://doi.org/10.1145/3160489.3160492
16. Hieke Keuning, Johan Jeuring, and Bastiaan Heeren, A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Transactions on Computing Education, 19(1), 2018.

17. Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart. 2013. A review of AI-supported tutoring approaches for learning programming. In Advanced Computational Methods for Knowledge Engineering. Springer, 267–279.
18. Thomas W. Price, Neil C.C. Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. 2016. Evaluation of a Frame-based Programming Editor. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16). Association for Computing Machinery, New York, NY, USA, 33–42. DOI:https://doi.org/10.1145/2960310.2960319
19. D. Hooshyar, R.B. Ahmad, M. Yousefi, F.D. Yusop & S.-J. Horng. (2015) A flowchart-based intelligent tutoring system for improving problem-solving skills of novice programmers. Journal of Computer Assisted Learning. 31(4): 345-361. doi: 10.1111/jcal.12099
20. Scott, A., Watkins, M. and McPhee, D. (2008) 'E-learning for novice programmers – a dynamic visualization and problem solving tool', 3rd Int. Conf. Information and Communication Technologies: From Theory to Applications, ICTTA, 7–11 April, Damascus, Syria, pp.1–6.
21. Gegg-Harrison T. S.: Exploiting Program Schemata in a Prolog Tutoring System. Phd Thesis, Duke University, Durham, North Carolina 27708-0129, (1993)
22. Dinesha Weragama and Jim Reye. 2014. Analysing student programs in the PHP intelligent tutoring system. Int. J. Artific. Intell. Edu. 24, 2 (2014), 162–188.
23. Budi Hartanto and Jim Reye. 2013. CSTutor: An intelligent tutoring system that supports natural learning. In Proceedings of the Conference on Computer Science Education Innovation and Technology. 19–26.
24. Edward Sykes. 2010. Design, development and evaluation of the java intelligent tutoring system. Technology, Instruction, Cognition and Learning 8, 1 (2010), 25–65.
25. Jay Holland, Antonija Mitrovic, and Brent Martin. 2009. J-LATTE: A constraint-based tutor for Java. In Proceedings of the Conference on Computers in Education. 142–146.
26. Sebastian Gross and Niels Pinkwart. 2015. Towards an integrative learning environment for java programming. In Proceedings of the IEEE Conference on Advanced Learning Technologies. 24–28.
27. Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2016. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. International Journal of Artificial Intelligence in Education (2016), 1–36.
28. Peter Brusilovsky1, Lauri Malmi, Roya Hosseini, Julio Guerra, Teemu Sirkiä and Kerttu Pollari-Malmi. (2018) An integrated practice system for learning programming in Python: design and evaluation. Research and Practice in Technology Enhanced Learning. 13:18 https://doi.org/10.1186/s41039-018-0085-9
29. Chen, S. and Morris, S. (2005) 'Iconic programming for flowcharts, Java, Turing, ETC', Conference on Innovation and Teaching Computer Science Education (ITiCSE), Caparica, Portugal, ACM, pp.104–107.