Improving Student Performance by Enforcing Algorithm Development

Jungsoon Yoo, Zhijiang Dong, Sung Yoo, Suk Seo, and Chrisila Pettay
Department of Computer Science
Middle Tennessee State University
United States
(csyoojp, zdong, cssung, sseo, cscbp)@mtsu.edu

Abstract: Having the ability to design algorithms for simple problems is one of the most important learning outcomes of introductory computer science courses. Unfortunately, teaching algorithm design has been a challenging task, and most students ignore this step due to a variety of reasons when they are solving problems. To address this challenge, we developed a web-based tutoring system to train students to develop algorithms prior to coding. In this paper, we present the results of experiments that were conducted to analyze the effectiveness of enforcing algorithm development prior to programming. The experimental results show that by using the web-based tutoring system, both the student overall performance and the student attitudes regarding the importance of algorithm development are improved. In addition, the students find the tutor to be helpful.

1. Introduction

A primary goal of introductory computer science courses is the development of program design and program construction skills using a computer programming language such as C++ or Java. In these courses, teachers introduce programming concepts, demonstrate how to solve a problem by developing an algorithm, and show how to implement an algorithm in a specific programming language. Typically, a programming assignment will be given for students to practice the skills learned in classroom lectures. These programming assignments are often so complex that most students need to design an algorithm before starting an implementation (programming). However, we observed that students typically skip the algorithm development phase, often with disastrous results, not the least of which is the increased time spent debugging. There appear to be two factors that contribute to this tendency to ignore the algorithm development phase – lack of appreciation and lack of ability.

The first factor is the lack of appreciation of the worth of the algorithm development phase. Students think it will take more time overall to develop an algorithm first. Thus, they start coding because they feel closer to the completion of the assignment. This view of algorithm development is wrong because with a correct algorithm, students will spend much less time on coding and debugging, and therefore reduce overall time spent on assignments. Unfortunately, such wrong impressions are frequently enhanced when the algorithms of these assignments are not explicitly graded. Many students even claim that algorithm development is not necessary to complete a programming assignment.

The second factor is the lack of ability to construct an algorithm from scratch. Students have difficulty understanding algorithm concepts and cannot write down their algorithms using natural languages other than programming languages. For the past few years, we have required students to turn in algorithms of programming assignments prior to program submission in introductory computer science courses. Even though we have demonstrated and used algorithms in classroom lectures, students still don’t understand how to develop an algorithm. Most of the turned in algorithms are a repeat or a rephrase of the problem specification provided in the assignment handout. Those that are not simply a repetition of the problem specification either are useless for implementing a program, or are completely wrong. Students also show an inability to apply the top-down approach; i.e., most of the students do not refine their initial algorithm.

In this paper, we describe the results of requiring students to use a system called AlgoTutor (Algorithm Tutor) to develop an algorithm prior to coding a programming assignment. We show that in introductory computer science courses the system helps improve the student performance on programming assignments and problem solving on written in-class exams. In the following section, we will describe the AlgoTutor system and briefly address the motivation behind its development. Section 3 describes the experiments we conducted in CS-II courses
and presents the results. Related work is given in Section 4. Conclusions and the future directions of the current project are presented in the last section.

2. The Algorithm Tutoring System

2.1. Motivation

Developing an algorithm prior to coding is important because it helps the students structure the programs and identify any issues they may encounter during problem solving. In the process of developing an algorithm, students may need help when they run into a problem that cannot be resolved on their own. However, there are several issues with manual tutoring for algorithm development:

• Timely feedback: Students may not receive their grade until the next class time; often they do not receive the grade before they start implementing their algorithms.
• Availability of help: Help is limited to when the teacher is available; however many times the student’s and the teacher’s schedules do not coincide, i.e., the student often needs help late at night.
• Usefulness of student algorithm: Based on our experience from previous years, many student algorithms are neither complete, correct, nor useful. This may explain why students don’t see the usefulness of developing an algorithm.
• Lack of algorithm development techniques: Even though teachers present algorithm development techniques such as top-down or bottom-up approaches and teach functional abstraction and data abstraction in class, students have difficulty applying these techniques to a specific problem solving situation.

These issues lead us to believe that we need a different approach to help students develop algorithms. Our solution, AlgoTutor is briefly described in the following subsections. For a more detailed description, see (Pettey et al. 2009).

2.2. System Description through an Example Problem

The AlgoTutor System addresses the aforementioned issues in various ways. The system is a web-based tutoring system so that students can access the system using a web-browser at their convenience. The system is interactive, i.e., the system provides online grading and feedback so that students can check immediately whether or not their algorithm is correct and what mistakes, if any, they made. The system also employs both the top-down and bottom-up approaches so that students are able to practice these techniques on various problems.

Using AlgoTutor, students develop an algorithm for a specific problem provided by the teacher using the following steps:

1. Operation selection phase: In this phase, for a given problem specification students compose an algorithm by selecting and ordering operations from a list of operations provided by the teacher.
2. Parameter mapping phase: In this phase, the students convert the abstract operations in the composed algorithm to problem specific operations by instantiating operation parameters.
3. Problem refinement phase: Some operations of an algorithm are complex enough to be defined as sub-problems requiring further refinement. In this phase, these operations are presented as sub-problems and students design algorithms for them by repeating the previous two steps.

In the rest of this section, we will explore the system in more details through an example problem. This problem is to write a program to simulate operating a taxi service: receiving customer requests, driving to customers’ locations, picking up customers, and delivering them to destinations. After serving each customer request, the program prints some statistical information such as the total distance traveled and the total gas consumed. The program gets the customer requests from an external file.
2.2.1. Operation Selection Phase

When a student starts the AlgoTutor system for a specific problem, the tutor will present a list of operations to be used for the algorithm along with the problem specification. (Tab. 1) shows the operation list for the taxi problem provided by AlgoTutor. The operations are divided into two groups: top level operations and lower level operations. The top level operations are the ones that are not part of any other operations. The lower level operations are ones that can be considered as detailed operations of the top-level operations. For example, the operation “Keep processing customer requests until all requests have been processed”, which is a loop control structure, is a top-level operation since it is not contained in any other operation in the final algorithm. The operation “Bool: More customer request available” is a lower level operation because it will be used as the logical condition of the above loop control structure.

A student creates an algorithm by dragging and dropping operations from the provided list of operations. There are buttons that enable a student to check the correctness of the working solution at any time. The system evaluates the solution against the correct solution(s) stored in the database. If the student’s algorithm is incorrect, the system points out the last correct operation in the student algorithm. This process can be repeated until the ordering of the student algorithm is correct or the student wants to move on to the next phase.

It should be noted that we have decided to provide a list of abstract operations for algorithm development instead of letting students use their own operations. The suggested operations are abstract operations that are used in the correct solution for the problem. There are two main reasons for this decision. First, students typically don’t know how to come up with an abstract operation. Secondly, many students don’t know where and how to start algorithm development. By providing operations, students will have a chance to see appropriate ways to form abstract operations and will be able to start developing an algorithm using suggested operations.

2.2.2. Parameter mapping phase

After operation ordering is done, the system presents the student with the parameter mapping phase. Parameter mapping is a process of changing a generic operation into a problem specific operation. For example “Create an object of a class” is a generic constructor operation, which has two parameters: the name of the object and the name of the class. By specifying the value for each parameter, an abstract operation is converted to a problem specific operation. For example, the operation “Create an object theCab of a class Vehicle” is a corresponding problem specific operation.

(Fig. 1) shows a snapshot of a parameter mapping phase. Students can replace the question marks (parameters of abstract operations) either by clicking on the question mark and selecting appropriate values from a pop-up menu or by dragging a variable tile from the right side to the question marks. Vertical bars around variables or literals indicate that the instructor locked those variables so the student has no choice. This eliminates the need for some alternative solutions and helps students figure out appropriate parameter mappings.
2.2.3. Problem Refinement Phase

A more complex problem has multiple levels of sub-problems. Each sub-problem is used as an atomic operation in the upper level. Solving each sub-problem can be considered as a step-wise refinement in the top-down approach. Solving a sub-problem prior to solving its parent problem is a form of the bottom-up approach as is reusing previously solved problems. For example, in the taxi problem, the operation “Drive the vehicle to (x, y) to drop off passengers” can be defined as a sub-problem so that students can be requested to design an algorithm for this operation by going through the three steps of operation selection, parameter mapping, and possibly refinement.

To enforce top-down and bottom-up design principles, complex problems are hierarchically structured into sub-problems. The top level problem and sub-problems can be solved in any order determined by the teacher so that it can reflect either the top-down or the bottom-up approaches.

3. Experiments

Our experiments have been designed to evaluate the effect of enforcing algorithm development prior to coding in introductory computer science courses. The experiments involved two instructors, each of them taught one or two CS-II sections during the 2008 spring and fall semesters, respectively. There were 62 students in Spring 2008 and 44 students in Fall 2008. Both instructors covered CS-II concepts in the same order, gave the same programming assignments, and used the same policies for both semesters. The experiments have measured the effects of using AlgoTutor prior to coding from three perspectives:

- Does using AlgoTutor prior to coding improve student performance on programming assignments?
- Does enforcing algorithm development in programming assignments improve problem solving skills?
- What is the students’ attitude toward using AlgoTutor?

Figure 1: The parameter mapping phase. A question mark is being replaced with the variable ifs.
3.1. Does Using AlgoTutor Prior to Coding Improve Student Performance on Programming Assignments?

Open lab assignments (OLAs) are small programming projects assigned to students so that they can solve problems at their own pace without direct teacher supervision. Typically, each OLA addresses a specific concept such as linked lists. Concepts of structured arrays, recursion, basic object-oriented programming, linear linked lists, stacks and queues are covered in OLA1 through OLA6, respectively. The problems used in OLAs for the same concepts are slightly different between semesters, but we believe that these problems are at the same difficulty level.

During Spring 2008, students were required to manually design algorithms from scratch before coding for the major functions for each of the OLAs. The algorithm contributes 20% to the OLA score, while the program contributes the other 80%. During Fall 2008, for the first three lab assignments students were given a set of algorithm design related questions to answer instead of submitting their algorithm design. For example, we created a set of algorithm design questions such as figuring out the base case(s) of a problem given for the recursion concept. For the rest of the labs, the students used AlgoTutor to develop the algorithms before implementation.

(Fig. 2) shows the comparison of the OLA submission rates for spring and fall semesters. The submission rates of Fall 2008 are better than the submission rates of Spring 2008 with an exception of OLA2. The average of all the OLA submission rates has gone up from 78.5% in Spring to 84.2% in Fall. The submission rates are the ratios of the number of OLA submissions to the number of the currently attending students. As an aside, the retention rate of spring was 79.2% and it was 83.7% for fall. The retention rates are calculated after excluding students who never took any exams.

(Fig. 3) shows the comparison of the average scores of OLAs for spring and fall semesters. The average scores of all OLAs in fall are better than those in spring. We would like to point out that the problems to solve for later OLAs usually are more complex because more advanced topics and concepts are covered in class as the semester progresses. This may explain why both the submission rates and the average scores decline for later OLAs.

\[\text{Figure 2: The comparison of the OLA submission rates for Spring and Fall 2008.}\]

\[\text{Figure 3: The comparison of the average scores of OLAs for Spring and Fall 2008.}\]

---

1 Tree concepts are covered in OLA7, however, the submission rate and scores are affected by the instructor’s policy and other factors. We have decided not to include OLA7 in this study.
To analyze the effect of using AlgoTutor before coding, we have grouped OLAs into two sets: 1. manual algorithm development both semesters (OLA1, OLA2, and OLA3) and manual algorithm development in the spring versus AlgoTutor algorithm development in the fall (OLA4, OLA5, and OLA6). This results in four groups: Spring-A (set 1 spring 08), Spring-B (set 2 spring 08), Fall-A (set 1 fall 08), and Fall-B (set 2 fall 08). (Tab. 2) shows the averages and standard deviations of these four groups. (Tab. 3) shows the ANOVA results. The following conclusions can be drawn:

- Even though there is a difference between the average scores of the manual algorithm developments in spring 2008 (Spring-A) and fall 2008 (Fall-A) it is not statistically significant. This means the difficulties of those problems are comparable between semesters.
- The performance improvement from Spring-B to Fall-B is statistically significant. This shows that using AlgoTutor prior to coding was effective enough to make a difference.
- Even though Spring-A and Spring-B are significantly different (probably because of the complexity difference for later OLAs), Fall-A and Fall-B are not significantly different. This may be the effect of using AlgoTutor, i.e., developing an algorithm prior to implementation helped students solve complex problems successfully.

Therefore, we conclude that using AlgoTutor prior to coding improved the student performance on programming assignments. In the next sub-section, we will analyze the students’ problem solving skills.
3.2. Does Enforcing Algorithm Development in Programming Assignments Improve Problem Solving Skills?

To check whether enforcing algorithm development in programming assignments improves problem solving skills, we have analyzed student answers on the final exams of these two semesters. The final exam questions that we examined are to write a code fragment or an algorithm in pseudo-code to perform a specific task. There was no identical question that appeared on all of the four final exams, but several questions had similar complexities on these exams. Since these questions are graded based on syntax as well as algorithm correctness, we have decided not to compare scores but to focus on their problem solving skills. Compared to the previous semester, the following are what we have observed from those students who used AlgoTutor in Fall 2008:

- The overall quality of student pseudo code has improved.
- More students were able to write correct pseudo-code. Even though some students did not receive full credit for their work, more students managed to provide reasonable pseudo-code.
- There were fewer students who left the answer completely empty.

Based on the above observations and our individual interactions with students, we claim that enforcing algorithm development improved student problem solving skills. Since the questions we used for analysis from final exams are to write code for simple problems, the problem solving skills of using top-down or bottom-up approaches could not be analyzed.

3.3. What is the Students’ Attitude toward AlgoTutor?

At the end of the 2008 fall semester, we conducted a survey to obtain student feedback for the AlgoTutor system. (Tab. 4) shows a part of the questionnaire used in the exit survey. The response to these questions may reflect how students feel about AlgoTutor’s ability to help with algorithm development. Participating in the exit survey was voluntary while the programming assignments and the final exam are mandatory since they are a part of the course requirements. A total of 24 students participated in the exit survey, and the student responses are summarized in (Tab. 5). It shows that 88% of the students agreed or strongly agreed with “The algorithm tutor helped you organize your thoughts and develop an algorithm.” It also shows at least 80% of students agreed or strongly agreed that “Using the algorithm tutor helped you spend less time coding,” and “You have a good understanding of the algorithm you developed using the algorithm tutor.” “It helped you understand the process of developing an algorithm” was agreed or strongly agreed by 75% of the students.

<table>
<thead>
<tr>
<th>1. The algorithm tutor helped you organize your thoughts and develop an algorithm.</th>
<th>Strongly agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. You have a good understanding of the algorithm you developed using the algorithm tutor.</td>
<td>Strongly agree</td>
<td>Agree</td>
<td>Neutral</td>
<td>Disagree</td>
<td>Strongly disagree</td>
</tr>
<tr>
<td>3. Using the algorithm tutor helped you spend less time coding.</td>
<td>Strongly agree</td>
<td>Agree</td>
<td>Neutral</td>
<td>Disagree</td>
<td>Strongly disagree</td>
</tr>
<tr>
<td>4. The algorithm tutor helped you understand the process of developing an algorithm.</td>
<td>Strongly agree</td>
<td>Agree</td>
<td>Neutral</td>
<td>Disagree</td>
<td>Strongly disagree</td>
</tr>
</tbody>
</table>

Table 4: A part of the exit survey questionnaire.

<table>
<thead>
<tr>
<th>Question 1</th>
<th>Question 2</th>
<th>Question 3</th>
<th>Question 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly agree</td>
<td>38%</td>
<td>33%</td>
<td>42%</td>
</tr>
<tr>
<td>Agree</td>
<td>50%</td>
<td>50%</td>
<td>38%</td>
</tr>
<tr>
<td>Neutral</td>
<td>4%</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>Disagree</td>
<td>4%</td>
<td>4%</td>
<td>8%</td>
</tr>
<tr>
<td>Strongly Disagree</td>
<td>4%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5: Students’ Answer to the Survey.
The exit survey also asked students to provide any comments related to their experience with AlgoTutor. The following are typical comments provided by students:

- Relatively easy to use. Good for developing an algorithm.
- It is forcing me to think which I liked.
- Helpful in learning how to develop an algorithm.
- Made it easy to think about a complex problem, the tutor makes it easy to see a large problem as a series of small steps.
- Focus on the greater perspective and concept behind the project.
- Helped with understanding overall algorithm.
- Easy to understand, it's very easy to read and understand. I thought it was very helpful.
- It attempts to separate algorithm design from concrete development.
- It gives me a sense of direction and assists me in developing an efficient algorithm.
- Provides a skeleton algorithm for students to use to follow smooth transition into algorithm design.

There were some constructive criticisms as well as complaints that will be useful to improve the system. Most of the personal conversations with students were very positive and we are quite encouraged to know that students were happy to use the AlgoTutor system before they start working on program implementation.

4. Related Work and Discussions

Being able to develop an effective computer algorithm to solve a problem is one of the most important learning outcomes in a computer science curriculum. Many studies have been done to improve student performance in computer science courses, especially CS-I and CS-II, which have high dropout and failure rates nationwide (Kinnunen & Malmi 2006).

One approach is to change the way students learn – besides learning from teachers in the classroom, students can learn from their peers, or classmates, without the immediate intervention of an instructor. Studies (Gehringer et al. 2005) have shown that peer learning and collaboration promotes greater conceptual and procedural gains for students, accommodates a broad range of learning styles, results in greater enjoyment of the learning task, and encourages a stronger persistence in learning.

Another approach to improve student performance in computer science courses is to introduce pedagogical tools to help students understand concepts, and provide practice opportunities to sharpen their programming skills. The tools can be Intelligent Tutoring Systems (ITS) (Schulze et al. 2000, Shute & Psotka 1996), or web-based tutoring systems (Arnow & Barshay 1999, Shah & Kumar 2002, Yoo et al. 2006). To help students in Computer Science learn problem solving techniques, several tools have been developed, such as ALVIS Live! (Hundhausen & Brown 2007), “Iconic Programmer” (Chen & Morris 2005), FLINT (Ziegler & Crews 1999), RAPTOR (Carlisle et al. 2005), and TRAKLA2 (Korhonen et al. 2003). ALVIS Live! is an environment that assists students in understanding the dynamic behavior of computer algorithms by allowing students to compose pseudo-code algorithms and evaluate each line of the algorithm code. Unlike ALVIS, “Iconic Programmer”, FLINT, and RAPTOR use flowcharts to compose algorithms. TRAKLA2, a web-based, computer-aided learning environment that helps teach algorithms and data structures, assesses a student’s understanding of a given algorithm by observing the sequence of transitions.

The tutoring systems mentioned above include some useful aspects that help Computer Science Students learn problem solving techniques or programming concepts, but none of the systems contains all of the features that are needed to train the students to practice algorithm development as we desire. AlgoTutor, the system that we are developing, is a web-based, interactive tool that allows students to practice both the top-down and bottom-up algorithm design methodologies by steering them through the process of designing a solution in a structured manner.

5. Conclusions and Future Work

In introductory computer science courses, the importance of the algorithm development is emphasized by enforcing algorithm development prior to coding for programming assignments. To this end a web-based tutoring system, called AlgoTutor, has been developed to allow students to practice developing algorithms. Students can use AlgoTutor for open lab programming assignments as well as for in-class programming exercises. AlgoTutor is
interactive and employs both the top-down and bottom-up approaches so that students will be able to practice these techniques on their own. The system provides opportunity for students to practice using terminology that is typically used in pseudo code.

We conducted experiments to analyze the effectiveness of enforcing algorithm development prior to program implementation. Our experimental results show that using AlgoTutor prior to coding does improve the performance on the programming assignments, and students have shown strong interest in the system and provided positive feedback. We believe that the system helped students by improving the quality of programs and by encouraging good problem solving skills.

The experiments conducted in spring and fall 2008 provided us valuable lessons. We will be revising AlgoTutor with the experience we gained during these semesters. A key point enforced by the experiments is that the analysis phase is a critical part of the problem solving process. That is, students cannot develop algorithms for problems that they do not understand. We are in the process of adding an interface to provide warm up questions that address problem specific algorithm development questions. Incorporating peer learning into AlgoTutor is another possibility for future work.

References


Acknowledgments

This research was supported by the NSF CCLI grant DUE-0737257.