Teaching Programming Concepts Using Algorithm Tutor

Jungsoon Yoo, Chrisila Pettey, Suk Seo, Sung Yoo
Computer Science Department
Middle Tennessee State University
United States
(csyoojp, cscbp, sseo, cssung)@mtsu.edu

Abstract: The Algorithm Tutor, AlgoTutor, is a pedagogical tool that encourages students to start the program solving process with algorithm design. The algorithm tracer in AlgoTutor teaches students to trace the flow of simple algorithms. Additionally, the ProgramPad component of AlgoTutor helps students implement the designed algorithm in C++ code which can be tested within the system. Using AlgoTutor, students can practice algorithm development and implementation skills in a web-based, interactive environment. The system provides online feedback for their algorithm so that students can work at their own pace at a convenient time. We have shown that using AlgoTutor to introduce programming concepts can significantly improve student comprehension of those concepts. We have also shown that using AlgoTutor as a reinforcement tool can significantly improve student comprehension of those concepts.

1. Introduction
In a typical introductory computer science class, programming concepts are taught by lecturing on the concepts and by showing how to solve example problems using the concepts. Programming assignments are used for students to practice on the learned concepts, while in-class examinations and program grades are used as assessment tools. However, it is well known that computer programming skills are considered difficult to develop and computer science courses have low retention rates (Fesakis and Serafeim, 2009).

In recent years, teaching tools with multimedia platforms such as Alice (http://www.alice.org) and Scratch (http://scratch.mit.edu) have become popular for introducing programming concepts. Compared to learning how to write programs in a formal programming language such as C++, learning how to use these tools is relatively easy, especially for the Nintendo generation (Guzdial and Soloway, 2002). An additional merit of these teaching tools is that understanding control structures such as iteration is much easier because the obscure syntax restrictions of traditional programming languages is absent. Instead, programming in these multimedia platforms is done by snapping together “programming blocks” to create programs.

Alice actually was an inspiration for our algorithm development system – AlgoTutor (Pettey et al., 2009, Yoo et al, 2009) but instead of using programming blocks, the goal of AlgoTutor is to guide students in designing a correct algorithm using abstract “operation blocks.” While Alice (Cooper et al., 2000) and Scratch (Fesakis and Serafeim, 2009; Resnick et al., 2009) grammar is limited to programming graphical objects, each abstract operation block in AlgoTutor can range from a single C++ instruction to a complex set of operations such as a C++ function. This flexibility allows AlgoTutor to teach programming concepts in both top-down and bottom-up manners by guiding students to design a correct solution without mastering the syntax of a programming language.

In this paper, we show that the AlgoTutor system can be used not only to introduce a new concept prior to lecturing, but also to reinforce a concept which has been introduced using a traditional method. In the following section, we discuss the usefulness of the AlgoTutor system in teaching programming concepts. In Section 3, we illustrate each component of the AlgoTutor system. Section 4 describes the experiments that we conducted in CS-I classes and presents the results. We wrap up with a discussion of the assessment and conclusions.

2. Teaching Programming Concepts in CS-I
The main reasons that multimedia platform systems such as Alice and Scratch are popular for introducing programming concepts are (1) it is easy to build a program because of a lack of formal language constraints, (2) the visual execution of the developed program is immediately available, and (3) it is engaging/entertaining for students to develop programs. However, these systems are limited to the visual domain, so they are not suitable for teaching non-visual problem solving such as mathematical calculations. In these instances, we need a more general purpose programming language. Typically when we use a general purpose language to teach a programming concept such as
iteration, we explain the components of the language construct, syntax, and semantics using examples. So the student immediately becomes involved in syntax issues instead of focusing on the programming concept. What we need is a system which is as easy to use as Alice, yet can be more general purpose. More importantly, it would be desirable to have a tutoring feature that could guide students to develop a correct algorithm for a given problem. In the following subsections, we will use iteration as an example programming concept that can be introduced using the system.

2.1 Issues with Iteration Concept
We typically introduce the iteration concept with the while statement. A common first example problem would be to “read and add 10 numbers and print the sum.” When asked to write an algorithm for the problem in pseudo-code, students often write as follows (see Tab 1).

<table>
<thead>
<tr>
<th>Student Algorithm 1</th>
<th>Student Algorithm 2</th>
<th>Student Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read 10 numbers</td>
<td>1. While counter is less than 10</td>
<td></td>
</tr>
<tr>
<td>2. Add 10 numbers</td>
<td>1.1 Read a number</td>
<td></td>
</tr>
<tr>
<td>3. Print sum</td>
<td>1.2 Add the number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3 Increment counter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Print sum</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Sample student algorithms with errors.

In Algorithm 1, the student has difficulty producing a step by step solution that a computer can follow. The algorithm does not contain a loop structure with the essential operations that are associated with a loop control variable. Algorithm 2 uses a loop control variable counter and sum, but fails to initialize the counter or the sum. In Algorithm 3, the loop test is incorrect and the printing sum step erroneously belongs to the loop body instead of being a step after the loop.

In order to produce a correct loop for a given problem, students need to figure out proper operations for various elements that are associated with the loop they are building. To do this they should think about questions like the following: What steps need to repeat? What should occur before or after the loop? What controls the start of the loop? What makes the loop stop?

Students should come up with essential steps that are associated with the loop control variable; initialization, test, and update. They should produce and place additional necessary steps in the correct places. They also should know how to keep track of the value changes of loop control variables and make sure their algorithm indeed performs a task that is required to solve a given problem. For a beginning programmer, however, coming up with all these steps can be overwhelming. In the next two subsections we outline the goals for a tool that will help students in the process of understanding a programming concept and then implementing a high level algorithm in code.

2.2 A Tool for Introducing a Programming Concept
It would be ideal if we could design appropriate exercises through which students could learn a programming concept in a lab environment. These lab exercises should

- be easy to use
- allow students to focus on the algorithm without worrying about syntax errors
- allow students to trace the execution and observe the value changes of different variables
- be able to check the correctness of students’ answers and allow them to correct their errors
- allow students to practice different problems with various levels of difficulties

Using these lab exercises, students could learn the programming concept by themselves before being exposed to the concept in classroom lectures. These lab exercises could also be used to reinforce the programming concept after students participate in lectures and complete programming exercises covering the given concept. In this case they could practice developing algorithms for more complex problems by carefully designing lab exercises.

2.3 A Tool for Associating High Level Algorithms with Implementations
However, developing an algorithm is only the beginning stage of writing a program. We have observed that some students have difficulty writing code even when they are given an algorithm. They do not make the connection between an algorithm in pseudo-code and a code fragment in a high-level language. It would be desirable to have a
system that could present these students a code fragment in a programming language which corresponds to a given algorithm. It would be especially helpful if those students could see some code fragment that has been converted from an algorithm that they have developed. In addition, the system would be very useful if it provided an environment where students were allowed to actually compile and execute the code. Dancik and Kumar have developed a system which uses templates to produce programs for count-controlled loops (Dancik and Kumar, 2003). But their system will not do general purpose concepts, nor will it show code that has been implemented from a student designed algorithm. In the next sections we describe the AlgoTutor tool and how we have used it to address the two issues of introducing a new concept and associating an algorithm with actual code.

3. Developing an Algorithm using AlgoTutor

AlgoTutor is a web-based tutoring system that allows algorithms to be constructed by assembling pre-defined operations and control structures via a graphical drag-and-drop interface. The pre-defined operations, the building blocks of the AlgoTutor system, are created by teachers who design problems and their algorithm solutions. Teachers can sometimes create/present these problems targeting students who have not been exposed to a programming concept. Other times teachers may be targeting students who solve problems to reinforce the concept that they have already learned in class. In this section, we will describe three of AlgoTutor’s components and how they can be used to introduce a programming concept and/or to reinforce the concept or to associate algorithms with code. The three components are the algorithm composer, the algorithm visual tracer, and ProgramPad.

3.1 Algorithm Composer

The main component of the AlgoTutor system is the algorithm composer that helps students develop an algorithm for a given problem. Figure 1 shows a snapshot of the algorithm composer that is used to solve a problem addressing a sentinel-controlled loop. The composer displays the problem description in the top pane, the available operations and variables are presented in the right pane, and a working algorithm is shown in the left pane. An algorithm is composed by dragging operation tiles and variable tiles from the right pane.

![Figure 1: A snapshot of algorithm composer for a sentinel-controlled loop.](image)

In AlgoTutor, an operation can be a generic operation such as a while-loop which can be used as a problem specific operation such as “while the number read is not 0, increment count.” This allows students to identify the tile that they want to use in the right operation pane. When they drop the tile in the solution pane, they realize that the operation is a while loop which contains two parts: the condition and the repetition part. This is done by the
teacher who creates the solution algorithm from the generic pool of operations and refines operations used in the algorithm to make them more problem specific so that students can figure out how to develop the algorithm using the list of operations. To address the top-down problem solving strategy, we separate the top level operations from the rest, which also helps students identify the tiles they want to use.

AlgoTutor can present a partially completed algorithm to students so that an exercise can focus on a specific concept rather than asking students to create the entire algorithm. In Figure 1 most of the operations are “Locked” which means those operations are a part of the partially completed algorithm and are not changeable. This problem is the first exercise that addresses the condition of a sentinel controlled while loop.

In the process of developing an algorithm, students may need help when they run into a problem that cannot be resolved on their own. The AlgoTutor 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. When the “Compare my ordering w/ teacher’s” button is clicked, the system will highlight the last correct operation tile in blue and the first incorrect operation in red so that a student can fix the error and complete the algorithm. AlgoTutor can point out whether the student’s algorithm is correct or not (by matching with teacher’s solution), but it does not explain why it is incorrect.

3.2 Visual Algorithm Tracer

Given a list of operations and variables, students can construct an algorithm with the help of online feedback. However, following the execution of the algorithm to check for its correctness is not an easy task for many students in CS-I (Shackelford, 1997). We have developed an algorithm tracer that allows students to verify the correctness of their algorithm or to locate possible errors by providing visual tracing of the student algorithm. The visual algorithm tracer in AlgoTutor helps students gain a better understanding of algorithm execution.

To run the algorithm tracer, the student first selects a test input from the combo box located adjacent to the “Trace execution” button, as shown in Figure 1. A trace may require values for input parameters and/or input, the student can provide their own values or use one of the test data sets provided by the teacher. The execution tracer executes the algorithm by stepping one operation or control structure part at a time. The student can follow the control flow and monitor changes in variable values.

![Figure 2: A snapshot of algorithm tracer for the algorithm in Figure 1.](image_url)

Figure 2 shows a snapshot of algorithm tracer for the algorithm that solves the problem of “reading numbers until zero is read using a loop.” The tracer interface is divided into four panes: the algorithm pane, the variable pane, the output pane and the control pane. The algorithm pane shows a listing of the current algorithm in a format similar to that of the composer. Operations are numbered for reference during execution of the algorithm. The next operation to be executed is highlighted as shown in Figure 2 (step 3). The variable pane shows a listing of all currently defined local variables with their current values. The last operation step that modified the value of that variable is shown in parenthesis to the right of the value. (For input problem parameters if they exist, “input ” is shown instead.) The output pane shows output from the execution as it progresses.

Students may submit their algorithms for grading by clicking the “Submit for grading” button. The student’s algorithm is then run against a known correct solution for all predefined test cases. The outputs for each test case are compared and the student receives a score that is the percentage of all test cases that matched. These grading runs
are performed without visualization, i.e. no diagnostic information will be provided. The student may submit any number of times, with only the last submission used for final scoring. Students can also execute the algorithm with selected data to see the output of the execution without tracing by clicking the “Execute w/ selected” button. Currently, the algorithm tracer can only be used for a certain category of problems whose algorithms consist of traceable operations. Traceable operations are implemented so that the execution can be visualized. We are planning to expand the set of traceable operation in the future.

3.3 ProgramPad

When students in CS-I are asked to write a program from an algorithm they have developed, they often do not see the relationship between the algorithm and the code. For this reason, we have developed another component of AlgoTutor called ProgramPad. ProgramPad converts most of the operations in the algorithm into C++ code, and provides an interface for completing the program. By observing the actual conversions in ProgramPad, students can better understand the connections between the algorithm that is developed using the composer and the program that is converted from the algorithm. Figure 3 shows a snapshot of ProgramPad for the algorithm that solves the problem of “reading numbers until zero is read using a loop.” The ProgramPad Interface is divided into four panes: the problem description pane, the program pane, the variable pane, and the control panel.

ProgramPad displays the problem description in the top pane. The problem parameters and local variables are displayed in the variable pane on the right. The program pane on the left is used to either convert the student algorithm developed in the composer into C++ code or to load the previously saved code. Along with the capability of converting the algorithm into C++ code, ProgramPad provides most of a simple IDE’s (integrated development environment) capabilities: edit, save, load, build, run, and print. The control panel displays the buttons related to these features.

![ProgramPad Interface](image)

**Figure 3:** A snapshot of ProgramPad for the algorithm in Figure 2.
An algorithm created by AlgoTutor does not contain variable declarations. As shown in Figure 3, converting the algorithm in Figure 1 into a program is complete except for the variable declarations and input values for testing the program. All variables used in the algorithm are listed in the variable pane, and students can simply decide what should be the data type for each variable. To test this program, values for the input stream should be provided.

Repeatedly practicing on problems similar to the above problem using ProgramPad will help students learn the programming concepts in addition to understanding the connection between the algorithm and the corresponding program. Unlike the algorithm composer, ProgramPad does not provide automatic grading. Students can complete their program which is converted from the algorithm and test the program using their own input values.

4. Experimental Results
The primary purpose of AlgoTutor is to aid students in understanding the various aspects of algorithm development. One of these aspects is understanding the operation of and appropriate use of each of the core programming constructs – sequence, condition, iteration, and subprogram. In order to evaluate AlgoTutor’s usefulness in this area, we devised two experiments. The first was to evaluate the efficacy of using AlgoTutor to introduce new concepts. The second focused on using AlgoTutor to reinforce a concept which had been introduced in lecture and lab. For both experiments students took a pretest, then did the AlgoTutor experiments, then took the posttest which was identical to the pretest. For both of these experiments, we chose the concept of iteration using the while-loop construct.

For Experiment 1 the students had not been exposed to the concept of iteration either in lecture or in lab. Therefore, the pre/posttest (shown in Figure 4) used only high level algorithmic concepts as opposed to containing C++ code.

1) The following algorithm is supposed to read a number then print out all the integers from 0 up to, but not including that number. Fill in the blank so that the algorithm will be correct.

1. Read a value into the variable stopNumber
2. print numbers starting from 0 to the stopNumber -1
   counter = 0
   while __________________________ is true, do
      2.2.1 print counter
      2.2.2 add 1 to counter

2) The following algorithm is supposed to read numbers and echo print them until the number read is -1. Fill in the blank so that the algorithm will be correct.

1. __________________________
2. while __________________________ is true, do
   2.1 print numberRead
   2.2 Read a value into the variable numberRead

3) Write an algorithm that will read and sum 10 numbers then print the answer.
   For example, if input values are 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, your algorithm should print 10.
   If input values are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, it should print 55.

Figure 4: Pre/Posttest for Experiment 1, “AlgoTutor’s Ability to Introduce New Concepts”

Students were given 15 minutes to take the pretest, and were assured that we were simply getting a baseline measurement of their current knowledge so they should not be concerned if they did not know the concepts in the pretest. Students were then given 45 minutes to do the AlgoTutor experiments. These experiments are summarized in Figure 5. For Experiment 1, the algorithm given to the students was partially complete. In particular, all the loop initialization and the condition “counter <= ?” were given. Students needed to determine only the value for the “?” and fill in the two statements inside the while loop. Exercise 2 was also partially complete. Students were given the loop initialization and update, and they had to fill in the condition and the loop action. For Exercise 3, students had to construct the entire algorithm. Exercise 4 was again a partial algorithm. Students were given the entire outer loop and the initialization of the inner loop. This left the condition, action, and update portions of the inner loop for the students to complete. At the end of 45 minutes, students were told to stop and take the posttest that, as mentioned before, was identical to the pretest. Eighty-one students participated in this experiment.
**Exercise 1:**
1170 while-loop: count-controlled while: **Partially filled solution**
Assuming two integer values m and n have been read (given),
1. Use the while loop to find the sum of integers m through n inclusive.
2. Display the resulting sum.

**Examples:**
For m = 1 and n = 5, the sum is 15 (1+2+3+4+5).
For m = 1 and n = 10, the sum is 55.
For m = 10 and n = 13, the sum is 46.

**Exercise 2:**
1170 while loop: sentinel controlled
Use a while loop to read numbers from input until a zero is read.
1. Count the number of the numbers read and print the result.
2. Do not count the zero as it is the end-of-input sentinel value.

**Examples**
If the input data contains 1 2 3 4 5 6 0, then it should print 6, not 7. It is possible to have 0 as the first value in the data file.

**Exercise 3:**
1170 while-loop: sentinel-controlled while
Assuming two integer values m and n have been read (given),
1. Use the while loop to print the multiples of m that are less than n in ascending order.
2. Do not assume that m is less than n.
3. Remember that any number is a multiple of itself.

**Examples:**
If m is 3 and n is 10, your algorithm should print 3 6 9.
If m is 5 and n is 10, your algorithm should print 5.
If m is 10 and n is 5, your algorithm should not print anything.

**Exercise 4:**
1170 while-loop: Nested while
Give an algorithm of a function that will output a right triangle.
The function has two input parameters: height: the height of the right triangle symbol: a character to be printed in the right triangle
Example output: for height = 5 and symbol = X
XX
XXX
XXXX
XXXXX
The base of the right triangle depends on the height.

---

**Figure 5:** AlgoTutor Exercises for Experiment 1.

It was clear from grading the pretests, that some students knew the concept already because they scored 100. If we eliminate those that knew the concept prior to the experiment, there were 77 subjects remaining. The results of the experiments are shown in (Tab 2). In analyzing the scores, it appeared that there was a significant improvement in scores between the pretest and the posttest, so we performed a two-tailed t-test for dependent samples with a result of $t = 5.771$ with 76 degrees of freedom. This shows that the students’ performance significantly improved with a confidence level of 95%.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Experiment 1 (77 students)</th>
<th>Experiment 2 (75 students)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std.Dev.</td>
</tr>
<tr>
<td>Pretest</td>
<td>31.8</td>
<td>26.4</td>
</tr>
<tr>
<td>Posttest</td>
<td>46.8</td>
<td>31.7</td>
</tr>
</tbody>
</table>

**Table 2:** Results of Experiment 1 and Experiment 2.

For Experiment 2, the students had already participated in Experiment 1 as well as having classroom lectures and closed labs on the while-loop construct. Since students were used to writing C++ statements at this point, the pre/posttest (shown in Figure 6) contained C++ code. Once again, students were given 15 minutes to take the pretest, 45 minutes to do the AlgoTutor experiments (summarized in Figure 7), and 15 minutes for the posttest. As before, the posttest was identical to the pretest. Eighty-two students participated in this experiment. It should be noted that, unlike Experiment 1, only Exercise 5 had a partial solution given to the students. Also, Exercises 3 and 6 were program pad exercises that corresponded to Exercises 2 and 5, respectively.

Once again, it was clear from grading the pretests, that some students knew the concept already because they scored 100. As before, we eliminated those that knew the concept prior to the experiment, and that left 75 students. The results of the experiments are also shown in (Tab 2). We performed a two-tailed t-test for dependent samples on this data with a result of $t = 4.650$ with 74 degrees of freedom. This shows that the students’ performance improved significantly with a confidence level of 95%.
(1-4) What is the output of each of the following?

1.```c
n = 0;
while (n <= 5) {  
cout << n << ' ';
n++;
}
```  
a. 0 1 2 3 4  
b. 1 2 3 4 5  
c. 0 1 2 3 4 5  
d. 1 2 3 4 5 6  
e. 0 0 0 forever

2.```c
count = 1;
while (count < 10)  
{
cout << "Hello";
count++;
}
```  
a. prints "Hello" 0 times  
b. prints "Hello" 8 times  
c. prints "Hello" 9 times  
d. prints "Hello" 10 times  
e. prints "Hello" forever

3.```c
int val = 1;
while ( val < 4 )  
{
val++;
cout << "value=" << val << endl;
}
```  

4. ```c
int i = 0;
while ( i < 3 )  
{
j = 5;
while(j < 7 )  
{
cout << "j=" << j << " " ;
j++ ;
}
cout << "i=" << i << " " ;
i++;
}
```  

5. Write a code segment that prints the odd numbers between 1 and 99 inclusive. Print one number on each line.

6. Write a nested while loop to print the following on 3 lines using cout << "*";

<table>
<thead>
<tr>
<th>Figure 6</th>
<th>Pre/Posttest for Experiment 2, “AlgoTutor as a Reinforcement Tool”</th>
</tr>
</thead>
</table>
| **Exercise 1:** Give an algorithm of a function that will output the even numbers between lower and upper backwards (in descending order). The function has two input parameters: lower and upper. The variable upper is an even number, and the variable lower is less than or equal to upper. Numbers should be on a line by themselves (each number followed by an end-of-line character).  
**Examples:** If lower=4 and upper=8, the output should be  
8  
6  
4  
If lower=5 and upper=10, the output should be  
10  
8  
6  |
| **Exercise 2:** Give an algorithm of a function that will output the even numbers between lower and upper forwards (in ascending order). The function has two input parameters: lower and upper. The variable lower is an even number, and the variable upper is greater than or equal to lower. Numbers should be on a line by themselves (each number followed by an end-of-line character).  
**Examples:** If lower=4 and upper=8, the output should be  
4  
6  
8  
If lower=4 and upper=7, the output should be  
4  
6  |
| **Exercise 4:** Read numbers from input until a zero is read. Count the number of negative numbers read and print the result. Do not count the zero as it is the end-of-input sentinel value.  
**Examples:**  
If the input data contains -1 -2 -3 -4 -5 -6 0, then it should print 6, not 7.  
If the input data contains 1 -2 3 -4 5 -6 0, then it should print 3.  
It is possible to have 0 as the first value in the data file.  |
| **Exercise 5:** Write the body of a function that will output the multiplications of numbers from 1*n to n*n in order. The function’s input parameter, n is the upper bound. On each line, the product of two numbers should be printed along with two numbers.  
**Examples:** If n is 3, the output should be  
1*1=1  
1*2=2  
1*3=3  
2*1=2  
2*2=4  
2*3=6  
3*1=3  
3*2=6  
3*3=9  |

| Figure 7 | AlgoTutor Exercises for Experiment 2. |
5. Survey Results

5.1 End-of-Exercise Online Survey
When students finished each exercise of the AlgoTutor Labs on the while loop concept, one or two questions were asked to collect students’ opinions on the components of AlgoTutor. Different questions were asked on the algorithm composer, the algorithm tracer, and ProgramPad depending on each exercise. (Tab 3) shows some of those questions and summarizes the responses of a total of 40 students on these questions (Participating in the online survey was voluntary). The actual survey questions have five options: strongly agree, agree, neutral, disagree, and strongly disagree. However, we merged strongly agree and agree as well as strongly disagree and disagree for simplicity. (Tab 3) shows that most of the students either agreed or strongly agreed with each of the question and very few students disagreed or strongly disagreed.

<table>
<thead>
<tr>
<th>Survey question</th>
<th>agree</th>
<th>neutral</th>
<th>disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The partial solution helped me figure out the nested while structure.</td>
<td>60.5%</td>
<td>27.9%</td>
<td>11.6%</td>
</tr>
<tr>
<td>The tracing feature helped me verify my algorithm.</td>
<td>74.5%</td>
<td>23.4%</td>
<td>2.1%</td>
</tr>
<tr>
<td>The new button, “Compare my ordering w/ teacher’s”, was useful and helped you develop an algorithm?</td>
<td>70.8%</td>
<td>27.1%</td>
<td>2.1%</td>
</tr>
<tr>
<td>The ProgramPad helped me understand the connection between the algorithm and the C++ code.</td>
<td>64.3%</td>
<td>33.3%</td>
<td>2.4%</td>
</tr>
<tr>
<td>Did the ProgramPad help you understand the process of developing a program using the algorithm for this problem?</td>
<td>75.9%</td>
<td>24.1%</td>
<td>0%</td>
</tr>
<tr>
<td>This exercise helped me learn how to check input errors using a while loop.</td>
<td>78.8%</td>
<td>18.2%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Table 3: A list of the end-of-exercise online survey questions and the survey results.

5.2 Exit Survey
At the end of the 2009 fall semester, we also conducted a written exit survey to obtain student feedback for the AlgoTutor system. (Tab. 4) shows a part of the questionnaire used in the exit survey for the algorithm composer and ProgramPad. Questions 1 through 6 are regarding the algorithm composer and Questions 7 through 9 are for ProgramPad. Participating in the exit survey was voluntary and a total of 74 students participated in the exit survey. The student responses are summarized in (Tab. 4). From that data we see that at least 87% of the students agreed or strongly agreed with “Selecting operations from the provided list helped you figure out the solution,” and “The process of choosing parameters helped you figure out the solution.” We also received favorable responses on ProgramPad.

<table>
<thead>
<tr>
<th>Survey question</th>
<th>agree</th>
<th>neutral</th>
<th>disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Not worrying about C++ syntax helped you concentrate on problem solving.</td>
<td>76%</td>
<td>19%</td>
<td>5%</td>
</tr>
<tr>
<td>2. Selecting operations from the provided list helped you figure out the solution.</td>
<td>87%</td>
<td>7%</td>
<td>5%</td>
</tr>
<tr>
<td>3. “Check ordering” button was helpful.</td>
<td>96%</td>
<td>3%</td>
<td>1%</td>
</tr>
<tr>
<td>4. The process of choosing parameters helped you figure out the solution.</td>
<td>91%</td>
<td>5%</td>
<td>4%</td>
</tr>
<tr>
<td>5. The algorithm tutor helped you organize your thoughts and develop an algorithm.</td>
<td>82%</td>
<td>9%</td>
<td>8%</td>
</tr>
<tr>
<td>6. The algorithm tutor helped you understand the process of developing an algorithm.</td>
<td>83%</td>
<td>7%</td>
<td>11%</td>
</tr>
<tr>
<td>7. Being able to see the C++ code that corresponds to my algorithm helped me understand the algorithm/program.</td>
<td>74%</td>
<td>18%</td>
<td>8%</td>
</tr>
<tr>
<td>8. Being able to execute the program for my algorithm and to see the output helped me understand the algorithm</td>
<td>80%</td>
<td>18%</td>
<td>2%</td>
</tr>
<tr>
<td>9. The ProgramPad helped you bridge the gap between the algorithm development and the implementation of the algorithm.</td>
<td>62%</td>
<td>24%</td>
<td>14%</td>
</tr>
</tbody>
</table>

Table 4: A part of the exit survey questionnaire and the survey results.

The exit survey also asked students to provide any comments related to their experience with AlgoTutor. The following are typical comments provided by students.

Comments on Algorithm Composer
- The execution tracing was helpful especially for understanding how to write effective loops.
• It was helpful in creating an algorithm. It helped coding the program step by step. The interface was clear.
• It really helped me organize my thoughts, and at times it provided an algorithm solution to the one I had in mind.
• We don't have to worry about typos and we can focus on solving the problem rather than stressing out over semi-colons and other syntax errors, and trace execution was very helpful.
• Helps to understand ordering of operations as is applicable to the program (especially with loops).
• It's very user friendly. The top level and bottom level parameters generally help. Trace the program is really useful.
• Easy to use, I especially liked the "check ordering" button.
• Drag and drop implemented very well, very clean interface.
• I like being able to use the mapping tools to see your design worked out.

Comments on ProgramPad
• I liked that it helped see how a program is made from an algorithm.
• I like seeing the AlgoTutor in action.
• I got to see my code (algorithm) at work.
• I like the ability to use AlgoTutor solution instead of writing program from scratch.
• It was a good way to get real coding experience without having to write the entire program yourself.
• You actually got to see the code and then you got to finish it.

There were some constructive criticisms that will be used to improve the system. Most of the personal conversations with students were very positive and we are quite encouraged to know that students were able to learn a new programming concept and reinforce their learning using the AlgoTutor system.

6. Conclusions
We have developed AlgoTutor for aiding student understanding of the process of algorithm development. This tool can be used for developing algorithms, tracing algorithms, and implementing high level algorithms with C++ code. Based on experimental results involving 81 students, AlgoTutor has been shown to result in significant improvement in student understanding of a programming concept when used as a tool for introducing the concept. Similar experiments involving 82 students showed that using AlgoTutor as a reinforcement tool for reinforcing programming concepts resulted in a significant improvement in student understanding of the targeted concept. In addition, it is clear from student feedback that they find AlgoTutor to be a helpful tool. Future experiments are planned to evaluate the effectiveness of the ProgramPad component in helping students make the connection between a high level algorithm and the actual program implementation.

7. Reference:


Fesakis, G. and Serafeim, K. (2009). Influence of the familiarization with "scratch" on future teachers' opinions and attitudes about programming and ICT in education Volume 41 , Issue 3 (September 2009), ITiCSE '09, Pages 258-262.


