A TUTOR FOR COUNTER-CONTROLLED LOOP CONCEPTS AND ITS EVALUATION

Garrett Dancik and Amruth Kumar

Abstract – We have developed a web-based tutor for teaching and testing counter-controlled loop concepts in C++. The tutor is designed to promote problem-based learning. It repeatedly generates problems, grades user’s answers and provides feedback about the correct answers. This paper describes the design of the tutor, outlines a test that we used to evaluate its effectiveness, and presents the results of the test. The test confirmed our hypothesis that using the tutor helps improve student learning. The improvement is statistically significant. This tutor can be used for practice or testing in Computer Science I.

Index Terms – Problem-Based Learning, Web-Based Tutors, Online Learning, Programming

1. INTRODUCTION

Counter-controlled loops, i.e., *for* loops provide a powerful mechanism for repetition in programming languages. An understanding of this topic is expected at the beginning computer science level, and mastery of this topic is essential for computer science students who wish to advance to higher-level programming concepts. Understanding *for* loops requires a comprehension of all of the components of the *for* loop – initialization, condition, update statements, and loop body – which may or may not always be present in a loop. In addition, counter-controlled loops can be classified as up-counting or down-counting, depending on the update statement, and can iterate zero or multiple times. In our experience, down-counting loops are more difficult for computer science students than up-counting loops, and zero-iteration loops confuse them. We have developed a tutor for *for* loops that addresses all these issues.

In programming, an understanding of theoretical concepts is often dependent on adequate exposure to several instances of program code. Education research suggests that specific experience is required to recognize general principles [15]. Problems are useful for providing specific experience in Computer Science, and problem-solving is known to improve learning [7]. Therefore, we developed a tutor that would help students learn for loop concepts by solving problems. Education research indicates that focused practice is just as important for learning as expansive projects [16]. Therefore, our tutor is designed to be used as a supplement to the traditional programming projects assigned in Computer Science I.

In this paper, we will describe the tutor that we have developed to promote problem-based learning [4] of *for* loops and the results we obtained from evaluating its usefulness. In Section 2, we will describe the design and features of our tutor. In Section 3, we will discuss the design of our test to evaluate the tutor and the results we obtained. In Section 4, we will discuss how our work relates to the work of others in this area. We will discuss future work in Section 5.

2. THE DESIGN OF THE TUTOR

Our tutor is designed to promote problem-based learning by giving the user the opportunity to repeatedly solve problems and learn from the feedback. The problems generated by the tutor target a user’s understanding of *for* loop execution, examining the output of such a loop and the number of times it iterates. The tutor has the following capabilities:

1. **Problem Generation:** It can automatically generate problems, and is capable of generating an unlimited number of problems.
2. **Grading and Feedback:** It can grade the user’s answer and provide detailed feedback about the correct answer.
3. **Statistics:** It can keep track of the user’s progress, including how many problems the user has attempted, solved correctly, and incorrectly for each type of problem.

We will describe problem generation and feedback next.

2.1 Problem Generation

Each problem consists of a C++ code segment involving a *for* loop. The user is asked to predict the output of the loop and the number of times it iterates. The tutor generates C++ code containing *for* loops based on predefined templates. An example of such a template is given in pseudo-BNF notation in Figure 1.
In the template, T signifies a data type, V a variable, and R a random number. The inequality contained in angle brackets following the ‘#’ provides constraints for the value of the random number. A variable, string, or an ‘endl’ following ‘<<’ is indicative of a cout statement. An ‘M’ or an ‘m’ preceding a variable or an ‘endl’ statement gives that statement a 50% probability of being generated. In the template in Figure 1, for example, the value of the variable <V1> will be output after the loop approximately 50% of the times that that template is used to generate a problem. An example of a code segment generated using the template is shown in Figure 2.

```
{  
    <T1><V1>=<R1>#<10<=R1<=15>;  
    for(<T1><V2>=<R2>#<1<=R2<=5>;  
        <V2>=<V1>;  
        <V2>=<V2>+<R3>#<1 <=R3<=3>) {  
           <<<V2><<mendl;  
        }  
        <<<MV1>;  
}
```

FIGURE 1
A TEMPLATE FOR AN UP-COUNTING FOR LOOP.

```c++
void project()
{
   int quantity = 14;
   for (int measure = 4;
        measure <= quantity;
        measure = measure + 3)
   {
      cout << measure << endl;
   }
   cout << quantity;
}
```

FIGURE 2
AN EXAMPLE OF A CODE SEGMENT GENERATED USING THE TEMPLATE SHOWN IN FIGURE 1.

We have currently devised 72 templates that generate the following types of loops: up-counting loops that use addition or multiplication to update the loop counter, and down-counting loops that use subtraction or division to update the loop counter. Depending on the loop condition, the above types of loops may result in zero or multiple iterations. The templates generate programmatic variations of these four types of loops.

The templates determine the structure of the C++ code that the tutor will generate. However, all variable and function names are randomly chosen from a pre-defined list, random numbers are generated based on the supplied constraints, and some output statements are randomly inserted into the code. This allows the tutor to generate an endless supply of problems, enabling it to focus on specific concepts and code structure without sacrificing diversity of code.

The tutor can randomly select a template for each problem or can cycle through a pre-defined list of templates in a specified order.

2.2 Feedback

The tutor can currently provide feedback at three levels:

- **No Feedback:** The tutor simply prompts the user to go on to the next problem. The tutor is configured to provide no feedback when it is used for testing.
- **Minimal Feedback:** The tutor states whether the user’s answer is correct or incorrect.
- **Detailed feedback:** In addition to whether the user’s answer is correct or incorrect, the tutor explains the correct answer by describing the behavior of the program as it is executed. This is the type of feedback that is referred to as demand feedback in traditional Intelligent Tutoring Systems literature [1], i.e., it is provided only when the user asks for it.

**Demand (Detailed) Feedback:** The feedback is designed to emphasize the various components of the for loop, including the initialization statement, the condition statement, the body of the for loop, and the update statement. Any code preceding or following the for loop is also explained. The complete explanation is assembled during the following stages of execution of the code, and is presented under these titles:

- **Before the for loop:** The tutor describes any variable declarations, assignments, or output statements that precede the for loop.
- **Initialization:** The tutor explains how the loop counter is initialized, either with a literal constant or a variable declared earlier.
- **Condition:** The tutor evaluates the condition of the loop – it identifies the operators being compared, and rewrites the condition with values for variables. Based on the result of evaluating the condition, the tutor indicates whether the loop is entered, exited, or not entered at all.
- **Loop Body:** The tutor explains the execution of all the statements within the body of the loop, such as output statements and variable declarations.
• **Update:** The tutor explains how the loop counter is updated and lists the new value of the counter.
• **After the loop:** The tutor explains the execution of any statements that follow the body of the loop.

“Before the loop” and initialization segments are explained only once at the start. The condition, loop body and update segments are explained for every iteration of the loop. “After the loop” statements are explained once after the loop is exited.

2.3 User Interface

Figure 3 shows the user interface of the tutor. The user is led through a clockwise flow of action: from the program code in the top left panel (1), to the problem statement and controls to input user’s answers in the top right panel (2), followed by the “Check My Answer” button in the middle of the right panel (not shown), the feedback in the bottom right panel (3), and finally, the “Next Problem” button at the bottom of the right panel (4).

The tutor makes the “Check My Answer” and “Next Problem” buttons available only in their correct contexts to avoid confusion. In its feedback, the tutor refers to the line numbers printed alongside the code in the top left panel. When providing feedback, the tutor color-codes the user’s answer with red if the answer is incorrect and green if the answer is correct.

3. EFFECTIVENESS OF USING THE TUTOR

We set out to test the following hypotheses for our tutor:
1. Using the tutor for practice will help improve student learning.
2. The tutor is more effective at helping students learn than exercises in a textbook.

We tested the tutor in three sections of *Computer Science I* course. The topic had already been covered in class in all three sections before the test. In this section, we will first describe the protocol we used for the test, and then discuss the results we obtained from the test.

3.1 The Protocol

We used a controlled test – we randomly divided the class into two groups – control and test groups. Each group answered a pre-test, practiced solving problems, and then answered a post-test. The only difference between the control group and the test group was the method of practice:
• The control group practiced with a workbook of problems that provided minimal feedback, viz., only the correct answers to the questions, as would be found in the exercise section of a textbook. Practice workbooks consisted of 50 problems that were also generated by the tutor. Answers to the questions were listed in the back of the workbooks, first for all the odd-numbered questions, and then for the even-numbered ones.
• The test group completed practice problems using the tutor, which provided detailed feedback as described in section 2.2. Students were seated at separate workstations.

We used the tutor to conduct the pre-test and post-test – the tutor was configured to provide no feedback during the tests. The tutor generated problems for the pre-test, practice session, and post-test based on the *same* predefined sequence of templates. The same sequence was also used for the problems in the printed workbook.

Finally, we tested the tutor in two stages – the first for up-counting loops and the second for down-counting loops. We used the pre-test-practice-post-test sequence in each stage. We used a crossover design overall – the control group in the up-counting stage served as the test group in the down-counting stage, and vice versa.

To summarize, the protocol was as follows:
• **Up-counting stage:**
  1. Pre-test – 5 minutes, using the tutor with no feedback.
  2. Practice – 8 minutes, control group with workbook, test group with tutor providing detailed feedback.
  3. Post-test – 5 minutes, using the tutor with no feedback.
• **Down-counting stage:**
  4. Repeat of steps 1-3, except that during practice, control and test groups were switched from the up-counting stage.
• **Feedback:** Both groups filled out a feedback form on the version of the tutor they had used for practice. The feedback form included questions on the usability, learnability, and usefulness of the tutor as a problem-solving tool for learning.

3.2 The Results

We compared the percentage of problems that students correctly solved on the pre-test and post-test. We analyzed student performance on predicting the output of loops separately from estimating the number of iterations. Tables I and II summarize the figures for answering the output of the code and estimating the number of iterations, respectively. Inconsistent N-values reflect students who did not attempt any problem in a given section.
FIGURE 3
THE CLOCKWISE FLOW OF ACTION IN THE TUTOR — PROBLEM TYPE IS UP-COUNTING, ADDITION, MULTIPLE ITERATIONS

TABLE I
IMPROVEMENT FROM PRE-TEST TO POST-TEST AMONG WORKBOOK AND TUTOR USERS FOR CORRECTLY ANSWERING THE OUTPUT OF THE CODE

<table>
<thead>
<tr>
<th></th>
<th>Workbook Users</th>
<th>Tutor Users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Up-counting</strong></td>
<td>N = 25</td>
<td>N = 24</td>
</tr>
<tr>
<td>Average</td>
<td>0.13</td>
<td>0.31</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.36</td>
<td>0.29</td>
</tr>
<tr>
<td>2-tailed p-value</td>
<td>0.08436</td>
<td>0.00003</td>
</tr>
<tr>
<td><strong>Down-counting</strong></td>
<td>N = 25</td>
<td>N = 22</td>
</tr>
<tr>
<td>Average</td>
<td>0.05</td>
<td>0.33</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.20</td>
<td>0.32</td>
</tr>
<tr>
<td>2-tailed p-value</td>
<td>0.18908</td>
<td>0.00007</td>
</tr>
</tbody>
</table>

Clearly, the improvement from pre-test to post-test was greater for tutor users than for workbook users. The improvement in scores was always statistically significant for tutor users ($p < 0.05$), and never so for workbook users.

TABLE II
IMPROVEMENT AMONG WORKBOOK AND TUTOR USERS FOR CORRECTLY ESTIMATING THE NUMBER OF LOOP ITERATIONS

<table>
<thead>
<tr>
<th></th>
<th>Workbook Users</th>
<th>Tutor Users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Up-counting</strong></td>
<td>N = 24</td>
<td>N = 21</td>
</tr>
<tr>
<td>Average</td>
<td>0.09</td>
<td>0.26</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.42</td>
<td>0.40</td>
</tr>
<tr>
<td>2-tailed p-value</td>
<td>0.29044</td>
<td>0.00794</td>
</tr>
<tr>
<td><strong>Down-counting</strong></td>
<td>N = 23</td>
<td>N = 23</td>
</tr>
<tr>
<td>Average</td>
<td>0.05</td>
<td>0.16</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.31</td>
<td>0.38</td>
</tr>
</tbody>
</table>
users. Within each group, the score was always better when using the tutor than when using the workbook.

In Table III, we have combined the corresponding results of up-counting and down-counting stages to provide an overall comparison between workbook and tutor users. Once again, the improvement among tutor users was greater than that among workbook users, and the results were statistically significant.

<table>
<thead>
<tr>
<th>TABLE III</th>
<th>OVERALL IMPROVEMENT AMONG WORKBOOK AND TUTOR USERS COMBINING UP-COUNTING AND DOWN-COUNTING PROBLEM SETS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Workbook Users</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>N = 50</td>
<td>N = 46</td>
</tr>
<tr>
<td>Average</td>
<td>0.09</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.29</td>
</tr>
<tr>
<td>2-tailed p-value</td>
<td>0.03028</td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
</tr>
<tr>
<td>N = 47</td>
<td>N = 44</td>
</tr>
<tr>
<td>Average</td>
<td>0.07</td>
</tr>
<tr>
<td>Std-Deviation</td>
<td>0.36</td>
</tr>
<tr>
<td>2-tailed p-value</td>
<td>0.20187</td>
</tr>
</tbody>
</table>

We are confident of ascribing the improvement to the use of the tutor because we had minimized extraneous influences – students did not take a break between tests, they did not have access to textbook or any other reference material during the test, and they were not allowed to discuss among themselves during the test.

4. RELATED WORK

Our tutor has advantages over textbooks, the traditional source of problems for students:
1. The tutor can provide detailed feedback unlike printed textbooks.
2. The tutor can generate an unlimited supply of problems, thereby providing as much practice with solving problems as the learner wants/needs.

Tutors such as ours have been developed for quantitative disciplines such as Physics (e.g., CAPA [8]), and electronics and control systems (e.g., CHARLIE [3]). Examples of such tutors developed for Computer Science include PILOT [5], SAIL [6] and Gateway Labs [2]. PILOT is a problem generation tool for graph algorithms, SAIL is a LaTeX-based scripting tool for problem generation, and Gateway Labs generate problems on mathematical foundations of Computer Science. In our work, we have attempted to build tutors for problems based on programs, problems for which the answers may not always be quantitative, e.g., pointers for indirect addressing in C++ [11] and dynamic memory management [12]; nested selection statements in C++ [18], static scope in Pascal, which permits nested scope declarations [13,14], parameter passing in programming languages [17], and classes and encapsulation in C++[10].

The use of problem generation systems has been shown to increase student performance by 10% in Physics [9], largely due to increased time spent on the task. Our evaluation seems to support this result.

Since our tutor is capable of generating new problems ad-infinum, instructors may use our tutor to assign homework or even administer tests without the fear of plagiarism. Since the tutor is delivered over the web, instructors may use it for distance education.

5. FUTURE WORK

We plan to extend this tutor to include problems of the following types:
- Loops wherein loop parameters are modified in the body of the loop, infinite loops, and nested loops
- Logic-controlled loops – while and do-while.

We also plan to extend the tutor to handle Java and C# loops. Finally, we plan to continue to test the tutor in future sections of our Computer Science I course.

The tutor is implemented as a Java applet so that it can be accessed over the Web without constraints of time and space. The tutor uses Swing classes, consists of 27 classes, and is about 471K in size. It is currently available over the Web at http://orion.ramapo.edu/~amruth/problets.

6. ACKNOWLEDGMENT

Partial support for this work was provided by the National Science Foundation’s Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864.

7. REFERENCES


November 5-8, 2003, Boulder, CO


