Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 224 (2009) 115–124
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Integrating test generation functionality into the Teaching Machine environment

Michael Bruce-Lockhart[1]

Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, Canada

Pilu Crescenzi[2]

Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Firenze, Italy

Theodore Norvell[3]

Electrical and Computer Engineering
Memorial University of Newfoundland
St. John's, Canada

Abstract

We propose an extension of the Teaching Machine project, called Quiz Generator, that allows instructors to produce assessment quizzes in the field of algorithm and data structures quite easily. This extension makes use of visualization techniques and is based on new features of the Teaching Machine that allow third-party visualizers to be added as plugins and for new scripting capabilities. Using these new capabilities, several quiz types have already been produced, which can be applied to any algorithm and/or data structure for which the necessary visualizer plugins exist.

Keywords: Algorithms and data structure, self-assessment, visualization, program animation

1 Introduction

Allowing students to test their knowledge in an autonomous and automatic way is certainly one of the most important topics within computer science education and distance learning. Indeed, many systems have been proposed in the literature

[1] Email: mpbl@mun.ca
[2] Email: pcrescenzi@unifi.it
[3] Email: theo@mun.ca

doi:10.1016/j.entcs.2008.12.055

for automatic assessment of exercises in the field of programming (e.g. [17,1]), in the field of algorithm and data structures (e.g. [12,11]), and in the field of object-oriented design (e.g. [7]). Two of the most important features that these systems should exhibit are, from the instructor's point of view, ease of use and, from the student's point of view, the possibility of replicating the same kind of test with different data. In this paper, we focus our attention on the automatic generation of assessment quizzes in the field of algorithms and data structures and on the use of visualization techniques in generating quizzes [6].

In order to keep the level of difficulty encountered by the instructor while generating a new kind of test reasonably low, we decided to avoid tests based on the manipulation of a data structure, such as the ones described in [10]. In particular, we focused our attention on a specific set multiple-choice quizzes; nonetheless, we think that the coverage of test types proposed in the following section is quite wide.

Our approach consists of adding quiz generation functionality to the existing Teaching Machine [4,5] and WebWriter++ [3,2] environment. The Teaching Machine is a program animation tool for visualizing how Java or C++ code runs on a computer. It contains compilers for the two languages and an interpreted run-time environment that runs on a pedagogical computer model that incorporates aspects of both the underlying machine (physical memory, fetch and execute cycles), the compiler (expression parsing) and the memory manager (a stack, static memory and a heap). It has always contained the capability for visualizations at a higher level of abstraction (e.g. a linked view of data) and has recently been extended to allow arbitrary visualizer plugins. The Teaching Machine is written in Java and may be run as an applet or as an application. WebWriter++ is a small authoring system written in JavaScript whose purpose is to allow authors of pedagogical web pages to focus on content rather than technology. It provides a number of other automated facilities such as displaying colour stained code in a visual container with buttons to execute the code in the teaching machine, edit the code, or play a video about it.

1.1 Structure of the paper

In the next section, we describe the five quiz types that are already included in our framework. In Section 3, we briefly describe the Teaching Machine and WebWriter++ extensions, which have been made in order to develop the *Quiz Generator* framework. These extensions mainly consist of a new plugin architecture and an enhanced scripting capability. In Sections 4 and 4.1, we describe a test example and how the new features of the Teaching Machine and the WebWriter++ tools allow the system to visualize and assess the test. Section 5 looks at related work. We conclude in Section 6 by listing some research questions concerning the possibility of using Quiz Generator as a testing tool, and not only as a self-assessment tool.

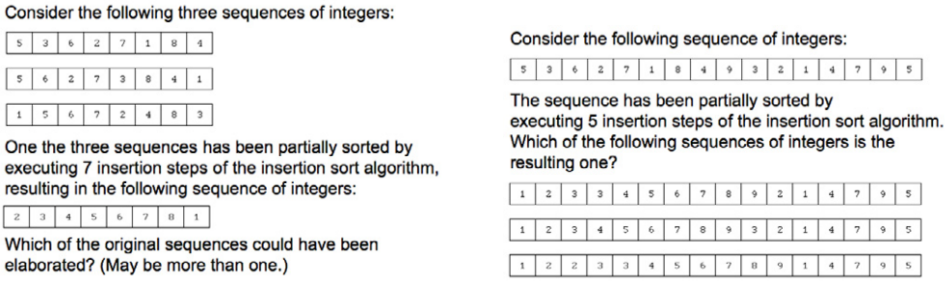

Fig. 1. The first and second test types: determining the correct input (left) and determining the correct output (right)

2 Test types

Our system supports several types of quiz question. These types are similar to the ones included in the taxonomy proposed in [9]: observe, however, that our main goal is to deal with multiple-choice quizzes, while the taxonomy is mainly oriented towards algorithm simulation exercises. In all the question types, visualizer plugins are used to produce both visualizations of the input and visualization of the output.

- Given a set of different inputs and a state S of a data structure, determine on which input the algorithm was executed in order to reach state S. For example, given a set of sequences of integers and given a partially sorted array A, the student is asked to determine which input sequence produced the array A after a specified number of sorting steps have been executed See left part of Figure 1.
- Given an input and a set of states of a data structure, determine which state has been produced by the (partial) application of the algorithm to the input. For example, given an input sequence of integers and three partially sorted arrays, the student is asked to determine which array corresponds to the execution of a specified number of steps of a specified sorting algorithm applied to the given input. See the right part of Figure 1.
- Given an input and the state S of a data structure, determine which algorithm has been used to produce the state after (partial) execution. For example, given a sequence of integers and a partially sorted array, the student is asked to determine which sorting algorithm has been applied in order to produce the partially sorted array. See Figure 2.
- Given an input and the state S of a data structure, determine how many "steps"

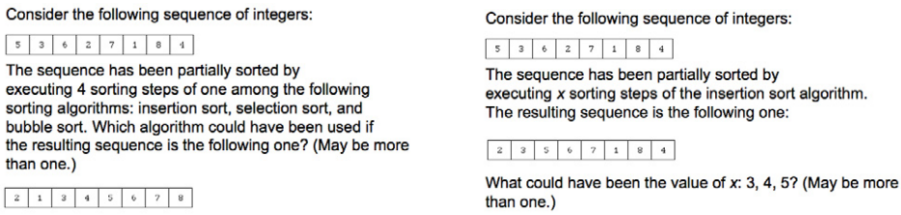

Fig. 2. The third and fourth test type: determining the algorithm (left) and determining the number of steps (right)

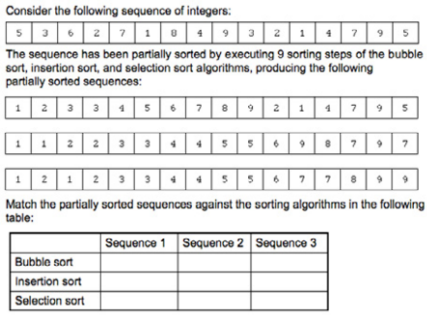

Fig. 3. The fifth test type: matching the output against the algorithm

have been executed by a specific algorithm to produce the state. For example, given a sequence of integers and a partially sorted array S, the student is asked to derive how many sorting steps have been executed by a specified sorting algorithm in order to transform the input sequence into the array S. See right part of Figure 2.

- Given an input and a set of states, determine which algorithms have been used in order to produce each of the states. For example, given a sequence of integers and given three partially sorted arrays, the student is asked to determine which sorting algorithm (among a specified set) produces each of the three arrays. See Figure 3.

Clearly, creating new test types depends quite heavily on the availability of specialized visualizers as well as the development of a means to capture their outputs at specific points.

3 System software architecture

The Quiz Generator project is an extension of the Teaching Machine project. As such it extends the two primary tools of this latter project — the Teaching Machine, which is a program animation tool written in Java, and WebWriter++, which is a JavaScript library for authoring interactive web pages for teacing and learning programming.

3.1 Visualization plugins

Quiz Generator leverages an extensive rewrite of the Teaching Machine carried out in 2006-2007, which permitted the development of third party plugins for the Teaching Machine. This development allows instructors to develop their own plugins without touching, or even recompiling, the Teaching Machine core. While such plugins are not confined to visualizers per se, we believed visualizers would be a core need. To that end a visualization adapter was developed to permit rapid third party development. The objective is to allow experienced developers to create new visualizers in a matter of between one and three days of programming. Indeed, it was the availability of this capability that got us thinking about developing a quiz

generator capability in the first place.

An important goal of our system is to allow the instructor to produce a test quite easily. The kinds of tests proposed require a number of different visualizer plugins which, even at only a day or two apiece, can add up quite significantly. Nevertheless, such development cannot be charged against test development as it would be unreasonable to present students with a visualization on a quiz that they had not seen in the course. Thus, for the purposes of this exercise, we assume that appropriate visualization plugins already exist and have been used in the course.

3.2 Scripting the Teaching Machine

Easy production also means a instructor should be able to produce a test without modifying the implementation of a data structure and/or of an algorithm. For example, if we refer to the first test type example and if we assume that the instructor has already programmed a Java or C++ class implementing a heap, then the test can be deployed without modifying this code by simply inserting a few scripting commands in the Java or C++ code, as comments, and by inserting a few JavaScript commands within the test web page.

The communication between the host web-page, the Teaching Machine, and the subject (Java or C++) code goes as follows.

(i) A JavaScript command within the web page invokes the execution of the Java or C++ code within the Teaching Machine.

(ii) Scripts, embedded as comments within the Java or C++ code, command the Teaching Machine to produce image files representing the state of one or more data structures.

(iii) JavaScript commands within the web page collect the image files and integrate this information within the question text.

This approach builds on earlier work with interactive learning pages which utilizes the connection between the WebWriter++ authoring tool and the Teaching Machine. Moreover, because the execution of the Java or C++ code is done wholly within the Teaching Machine, we have full control of this execution and of all the variables involved in the execution itself.

What was needed for the Quiz Generator project was a richer set of embedded scripting controls for the Teaching Machine than we had needed WebWriter++. For example, our learning web pages can currently display a code fragment for discussion, then allow a student to launch the example in the Teaching Machine to run it for herself, to edit it, or to possibly watch a video about it. Creating quizzes is more demanding.

4 A sample quiz

Here we expand on the example given for the third type of test described in Section 2 (see left part of Figure 2). Ideally, a student would be presented with a visualization

of an unsorted array, randomly populated according to parameters laid out by the instructor. A second snapshot of the array is presented after a partial sort, together with a list of algorithms. The student is told how many sorting passes were done and asked to check all algorithms that could have created the second snapshot.

Again, it is assumed that both the appropriate visualization plugins and an implementation of the sorting code and data structure already exist and have been used in the course.

To create the quiz, the instructor first instruments the code with testing parameters, for example:

(i) The size of the array (or a range of sizes, from which one size will be randomly picked).

(ii) The value range desired for random population of the array.

(iii) The sorting algorithm to be used.

(iv) The number of sorting passes (or, again a permissable range).

The code (or really code sets, since different pieces of code will be required for different topics) and the visualizations form a resource base for creating actual quizzes. The quizzes themselves are be created in HTML (or XHTML) using QuizWriter++, an extension to WebWriter++.

4.1 Scripting from inside and outside

Let us first consider the following simpler quiz question: given an unsorted array A and a snapshot of A after a specific sorting algorithm has been partially applied, the student is asked to determine how many sorting steps have been executed. In terms of controlling the Teaching Machine, this question is quite limited. We need to be able to

(i) Load the appropriate code into the Teaching Machine.

(ii) Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.

(iii) Start up the Teaching Machine to run invisibly (so the student cannot inspect it).

(iv) Specify the visualizer.

(v) Take snapshots of the data at appropriate points in the execution.

(vi) Recover the two snapshots (before and after) from the visualizer.

The first three and the last of these actions can be controlled via commands sent from the web page to the Teaching Machine. However taking snapshots is best controlled by script calls embedded within the subject code. These script calls call out to the Teaching Machine. It was necessary to develop a second scripting capability. To distinguish between them we call scripting from the JavaScript on the quiz page 'external scripting' and scripting from within the running code 'internal scripting'. Table 1 shows a number of potential scripting calls as well as their current

External scripting commands		
Command	*Effect*	*Status*
`run(filename)`	Loads filename into the Teaching Machine and waits at 1st line	Pre-existing
`autoRun(filename)`	Loads filename into Teaching Machine and runs it invisibly	Built
`insertPorthole(name)`	Create a container in the quiz for a snapshot	Built
`putSnaps()`	Load all snapshots from the Teaching Machine into portholes	Built
`addCLArg(arg)`	Add a command line argument for the program to be run in Teaching Machine	Built
Internal scripting commands		
Command	*Effect*	*Status*
`relay(id, call)`	Relay function call to plugin id	Built
`snapshot(id, name)`	Take a snapshot of plugin id for porthole name	Built
`stopAuto()`	Stop execution at this point	Built
`makeRef(id)`	Use the data structure in plugin id as a reference for comparison	Built
`compare(id)`	Compare data structure in plugin id to the reference data structure	Built
`returnResults`	Ship snapshot results back to quiz script	Built

Table 1
Scripting commands

status.

The quiz questions of Figures 1-3 were produced by using the capabilities of Table 1. For example, to engage fully the question posed at the beginning of Section 4 requires the following:

(i) Load the appropriate code into the Teaching Machine.

(ii) Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.

(iii) Start up the Teaching Machine to run invisibly (so the student cannot inspect it).

(iv) Specify the visualizer.

(v) Take a snapshot of the input state data.

(vi) Start the first algorithm

(vii) Have the Teaching Machine stop after it has executed the requisite number of sorting steps and take a snapshot.

(viii) Capture the output state.

(ix) Execute all other algorithms on the same input data.

(x) At the end of each other algorithm, have the visualizer compare the state of the array to that saved after the reference algorithm.

(xi) Recover the two snapshots (before and after) from the visualizer.

(xii) Recover data specifying algorithms that produced equivalent sorts.

(xiii) Build HTML for the quiz question.

The scripting calls in Table 1 were arrived at by examining just such quiz scenarios.

5 Related work

This paper fits into the third level (that is, the responding level) of the learner engagement taxonomy presented in [15]. As stated in the introduction, it tries to avoid some of the impediments listed in [13] and faced by instructors, while adopting visualization techniques, by making the development of new quizzes as easy as possible, and by integrating them within a unified framework, such as the one provided by WebWriter++ and the Teaching Machine. (By the way, [15] and [13] provide a good background for the research and development described in this paper, as well as test settings for evaluation.) Other papers deal with the development of interactive prediction facilities such as [8] and [14], where web-based tools are presented and evaluated, and [16],where a tool-independent approach is described.

6 Conclusion and further research

By constructing and examining quiz scenarios we are currently refining what capabilities we need in order to be able to achieve the kinds of quizzes laid out in Section 2. Nevertheless, the existing capabilities already span almost the entire space of controls needed, in the sense that they require almost all the structural extensions to the Teaching Machine that are needed. The additional scripting mostly requires the addition of new functions rather than fundamental changes to the Teaching Machine structure.

6.1 Research questions

Indeed, we are quite excited to have come this far. In the early days of scripting development it was by no means always certain that we would be able to achieve

all our objectives. Now that the design space is spanned we can focus on the development of the extra functionality required. Once that is done, it will allow us to concentrate on the research questions that are at the core to the whole endeavour of automated testing:

(i) Given a space of possible questions an instructor might want to ask in data structures and algorithms, can we build a quiz generator that does a reasonable job of spanning that space? That is, can an instructor use it to examine most of the issues he might want?

(ii) Even if we are successful in (i), can we produce enough variations in questions for the tool to be useful over a large number of uses? That is, can we produce enough different quizzes?

(iii) If we are successful in (ii), can we produce a set of quizzes that are reasonably equivalent? That is, would students taking different quizzes perceive that they had been treated fairly?

The last question, of course, moves beyond the realm of self-testing into the more vexing issue of testing for credit. That brings up a whole set of important issues such as quiz security and the proper gathering of quiz data. Nevertheless, until these three primary questions can be answered positively, there is no point in embarking upon these other issues. We are very hopeful that our current approach will be sufficiently successful to require these other issues to be tackled in the future.

References

[1] Ala-Mutka, K., *A survey of automated assessment approaches for programming assignments*, Computer Science Education **15** (2005), pp. 83–102.

[2] Bruce-Lockhart, M., *WebWriter++: A small authoring aid for programming*, in: *Proc. Newfoundland Electrical and Computer Engineering Conference*, 2001.

[3] Bruce-Lockhart, M. and T. S. Norvell, *Interactive embedded examples: a demonstration*, SIGCSE Bulletin **38** (2006), pp. 357–357.

[4] Bruce-Lockhart, M. P. and T. S. Norvell, *Developing mental models of computer programming interactively via the web*, in: *Frontiers in Education Conference - Global Engineering: Knowledge without Borders, Opportunities without Passports*, 2007, pp. 3–8.

[5] Bruce-Lockhart, M. P., T. S. Norvell and Y. Cotronis, *Program and algorithm visualization in engineering and physics*, Electronic Notes in Theoretical Computer Science **178** (2007), pp. 111–119.

[6] Cooper, M., *Algorithm visualization: The state of the field* (2007), master thesis at Virginia Polytechnic Institute and State University.

[7] Higgins, C., P. Symeonidis and A. Tsintsifas, *The marking system for coursemaster*, in: *Proc. 7th Annual Conference on Innovation and Technology in Computer Science Education*, 2002, pp. 46–50.

[8] Jarc, D. J., M. B. Feldman and R. S. Heller, *Assessing the benefits of interactive prediction using web-based algorithm animation courseware*, SIGCSE Bull. **32** (2000), pp. 377–381.

[9] Korhonen, A. and L. Malmi, *Taxonomy of visual algorithm simulation exercises*, in: *Proc. 3rd Program Visualization Workshop*, 2004, pp. 118–125.

[10] Krebs, M., T. Lauer, T. Ottmann and S. Trahasch, *Student-built algorithm visualizations for assessment: flexible generation, feedback and grading*, in: *Proc. 10th Annual Conference on Innovation and Technology in Computer Science Education*, 2005, pp. 281–285.

[11] Laakso, M., T. Salakoski, L. Grandell, X. Qiu, A. Korhonen and L. Malmi, *Multi-perspective study of novice learners adopting the visual algorithm simulation exercise system TRAKLA2*, Informatics in Education **4** (2005), pp. 49–68.

[12] Malmi, L., V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä and P. Sistali, *Visual algorithm simulation exercise system with automatic assessment: TRAKLA2*, Informatics in Education **3** (2004), pp. 267–288.

[13] Naps, T., S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. J. Ross, J. Anderson, R. Fleischer, M. Kuittinen and M. McNally, *Evaluating the educational impact of visualization*, in: *Working Group Reports from 8th Annual Conference on Innovation and Technology in Computer Science Education*, 2003, pp. 124–136.

[14] Naps, T. L., J. R. Eagan and L. L. Norton, *JHAVÉ—an environment to actively engage students in web-based algorithm visualizations*, SIGCSE Bull. **32** (2000), pp. 109–113.

[15] Naps, T. L., G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. A. Velázquez-Iturbide, *Exploring the role of visualization and engagement in computer science education*, in: *Working Group Reports from 7th Annual Conference on Innovation and Technology in Computer Science Education*, 2002, pp. 131–152.

[16] Rößling, G. and G. Häußge, *Towards tool-independent interaction support*, in: *Proc. 3rd International Program Visualization Workshop*, 2004, pp. 110–117.

[17] Vihtonen, E. and E. Ageenko, *VIOPE-computer supported environment for learning programming languages*, in: *Proc. Int. Symposium on Technologies of Information and Communication in Education for Engineering and Industry*, 2002, pp. 371–372.