Adding test generation to the Teaching Machine

M. Bruce-Lockhart, P. Crescenzi, T. Norvell

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 on new scripting capabilities. Using these new capabilities, five quiz types have already been produced, which can be applied to any algorithm and/or data structure for which the necessary visualizer plugins exist.

Categories and Subject Descriptors: K.3.1 [**Computer Uses in Education**]: Computer-assisted instruction; K.3.2 [**Computer and Information Science Education**]: Self-assessment; F.2.2 [**Nonnumerical Algorithms and Problems**]: Computation on discrete structures

General Terms: Algorithms, Human Factors

Additional Key Words and Phrases: Algorithms and data structure, self-assessment, visualization, program animation

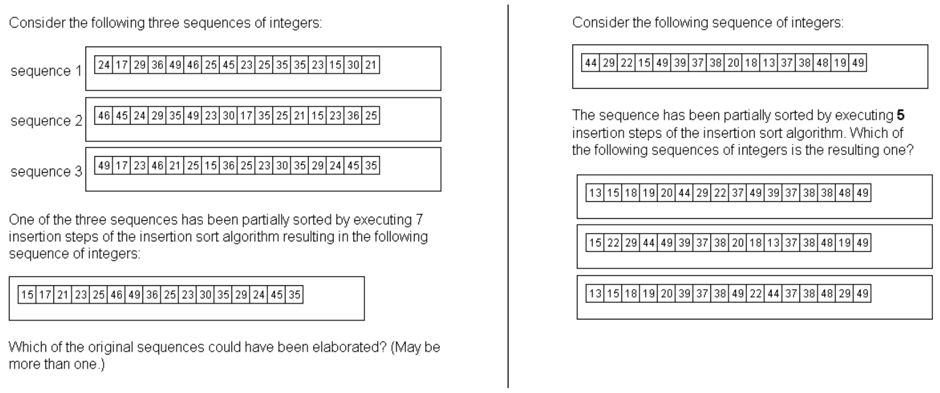
1. INTRODUCTION

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 for automatic assessment of exercises in the field of programming (e.g. [Vihtonen and Ageenko(2002); Ala-Mutka(2005)]), in the field of algorithm and data structures (e.g. [Malmi et al.(2004); Laakso et al.(2005)]), and in the field of object-oriented design (e.g. [Higgins et al.(2002)]). Two of the most important features that these systems should exhibit are, from the instructor point of view, ease of use and, from the student 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 [Cooper(2007)].

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 [Krebs et al.(2005)]. In particular, we focused our attention on a specific set of 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

A preliminary version of this paper [Bruce-Lockhart, Crescenzi, and Norvell(2009)] has been presented at the Fifth Program Visualization Workshop in Madrid.



Consider the following three sequences of integers:

sequence 1	24	17	29	36	49	46	25	45	23	25	35	35	23	15	30	21
sequence 2	46	45	24	29	35	49	23	30	17	35	25	21	15	23	36	25
sequence 3	49	17	23	46	21	25	15	36	25	23	30	35	29	24	45	35

One of the three sequences has been partially sorted by executing 7 insertion steps of the insertion sort algorithm resulting in the following sequence of integers:

15	17	21	23	25	46	49	36	25	23	30	35	29	24	45	35

Which of the original sequences could have been elaborated? (May be more than one.)

Consider the following sequence of integers:

44	29	22	15	49	39	37	38	20	18	13	37	38	48	19	49

The sequence has been partially sorted by executing **5** insertion steps of the insertion sort algorithm. Which of the following sequences of integers is the resulting one?

13	15	18	19	20	44	29	22	37	49	39	37	38	38	48	49
15	22	29	44	49	39	37	38	20	18	13	37	38	48	19	49
13	15	18	19	20	39	37	38	49	22	44	37	38	48	29	49

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

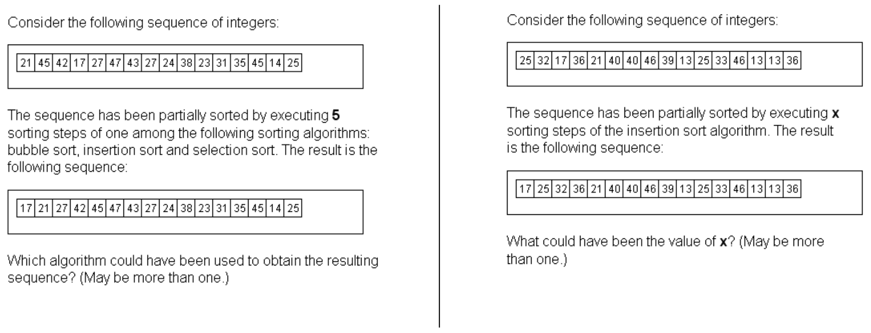
Teaching Machine environment [Bruce-Lockhart and Norvell(2007); Bruce-Lockhart et al.(2007)] and WebWriter++ [Bruce-Lockhart and Norvell(2006); Bruce-Lockhart (2001)]. The Teaching Machine is a program animation tool for visualizing how Java or C++ code runs on a computer. It contains compilers for the two languages and an interpreted run-time environment. The Teaching Machine presents the student with 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 had visualizations at higher level of abstractions (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 source code with syntax highlighting in a visual container decorated with buttons for executing the code in the Teaching Machine, editing the code, or playing a video demonstrating the example.

1.1 Test types

Our system currently supports the following five types of quiz questions. We use visualizer plugins to the Teaching Machine to produce visualizations of both input and outputs.

—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 the 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. (All figures in this paper are screen shots of web pages generated by our Quiz Generator system.)

—Given an input and a set of states of a data structure, determine which state



Consider the following sequence of integers:

21	45	42	17	27	47	43	27	24	38	23	31	35	45	14	25

The sequence has been partially sorted by executing **5** sorting steps of one among the following sorting algorithms: bubble sort, insertion sort and selection sort. The result is the following sequence:

17	21	27	42	45	47	43	27	24	38	23	31	35	45	14	25

Which algorithm could have been used to obtain the resulting sequence? (May be more than one.)

Consider the following sequence of integers:

25	32	17	36	21	40	40	46	39	13	25	33	46	13	13	36

The sequence has been partially sorted by executing **x** sorting steps of the insertion sort algorithm. The result is the following sequence:

17	25	32	36	21	40	40	46	39	13	25	33	46	13	13	36

What could have been the value of **x**? (May be more than one.)

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

has been produced by the (partial) application of the algorithm to the input. For example, given a sequence of integers and given 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 a state S of a data structure, determine which algorithm has been used to produce the state S after (partial) execution. For example, given a sequence of integers and given 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 the left part of Figure 2.

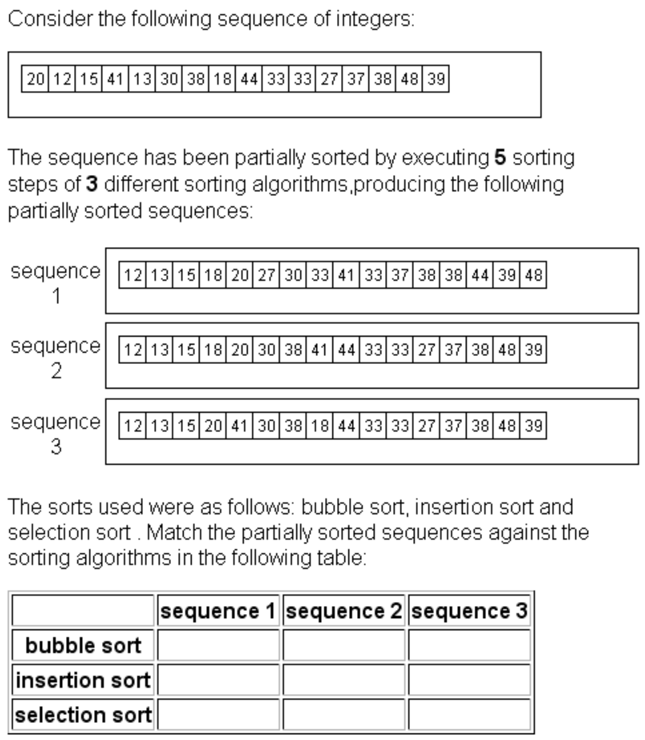
—Given an input and the state S of a data structure, determine how many "steps" have been executed by a specific algorithm to produce the state S. For example, given a sequence of integers and a partially sorted array S, the student is asked to derive how many sorting steps have been executed by a specified sorting algorithm in order to transform the input sequence into the array S. See the 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 suitable visualizers as well as the development of a means to capture their outputs at specific points. The default visualizers of the Teaching Machine show the states of all variables, and include a visualizer for linked structures that can show networks of objects such as linked lists and trees. The visualizer used to produce the visualization of arrays in Figures 1–3 was developed specifically for the Quiz Generator project.

1.2 System main characteristics

As we have already said, our test generation system is based on the Teaching Machine and WebWriter++ environment. While extending this environment in



Consider the following sequence of integers:

20	12	15	41	13	30	38	18	44	33	33	27	37	38	48	39

The sequence has been partially sorted by executing **5** sorting steps of **3** different sorting algorithms,producing the following partially sorted sequences:

sequence 1	12	13	15	18	20	27	30	33	41	33	37	38	38	44	39	48
sequence 2	12	13	15	18	20	30	38	41	44	33	33	27	37	38	48	39
sequence 3	12	13	15	20	41	30	38	18	44	33	33	27	37	38	48	39

The sorts used were as follows: bubble sort, insertion sort and selection sort . Match the partially sorted sequences against the sorting algorithms in the following table:

	sequence 1	sequence 2	sequence 3
bubble sort			
insertion sort			
selection sort			

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

order to add the quiz generation functionality, we have mainly taken into account the following characteristics.

Transparency. The quiz generation process should clearly be transparent to the final user, that is, the student. This implies not only that the Java or C++ code execution must be hidden from the student but also that the Java or C++ statements necessary to generate the quiz should not be shown, in case the instructor decides to allow the student to visualize the algorithm code (which is a feature already available in the original environment). To achieve this aim, we have introduced a new type of comment that is ignored by standard compilers and hidden by WebWriter++. On the other hand, the commented statements are compiled (and, hence, executed) by the Teaching Machine. This specially commented code is used to take snapshots of the visualization as the algorithm executes.

Visualization. Our quiz generator can make use of visualization techniques. Even though this feature is not strictly necessary, graphical presentation of a question can be a powerful aid to the student for understanding the question itself (for example, this should be especially true in the case of linked data structures such as lists and trees). To support this aim, our quiz generator has taken advantage of a feature recently added to the Teaching Machine. This is the possibility of extending the tool by means of plugins. In particular, a visualization adapter plugin has been implemented, so that producing a new visualizer plugin turns out to be a fairly straightforward task. For example, the simple visualizer used for Figures 1–3 took about two hours to create. Of course, using visualization techniques is not only for the purpose of quiz generation but also for teaching algorithm behaviour.

Ease of use. As already said, the task of generating a new quiz should be easy for the instructor. To achieve this aim, we have formally categorized the kind of tests that can be generated by our system and, for each category, we have developed several plugins that allow an instructor to make the generation of the quiz quite immediate. As we will see, the main difficulty in generating a new quiz in our systems is modifying the algorithm's implementation in order to take into account a specific execution parameter, such as the number of sorting steps executed. Clearly, since this parameter usually depends on the algorithm, its management cannot be, in general, fully automated.

Finally, we would also like to emphasize that our system can take advantage of all the features already available within the Teaching Machine and WebWriter++ environment. For instance, it is easy to add an immediate feedback functionality to our quiz generator that would allow the student to execute, step by step, an algorithm and view its execution, in order to understand why the given answer was not the right one or to understand why the given answer was indeed correct.

1.3 Structure of the paper

In the next section, we introduce the notion of "testable algorithm" and we formally define 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.

2. A CATEGORIZATION OF QUIZZES BASED ON TESTABLE ALGORITHMS

In this section we introduce the notion of "testable algorithm" and we propose a formal categorization of quizzes based on this notion. This allows us to determine the functionality that has to be added to the Teaching Machine and WebWriter++ environments in order to implement the quizzes themselves. Our categorization is similar to the taxonomy proposed in [Korhonen and Malmi(2004)]. However, our main goal is to classify multiple-choice quizzes, while their taxonomy was mainly oriented towards algorithm simulation exercises. Even though, as we will see, selecting the right answer usually requires the simulation of the execution of an algorithm, our categorization tries to emphasize the capability of a quiz to automatically generate wrong answers which appear reasonable to the student.

A testable algorithm is an algorithm whose output is determined not only by its input, but also by a an extra *execution parameter*. This extra parameter is generally not evident in the presentation of algorithm but is easily understood by the student. For example, in a sorting algorithm, the parameter might be a limit on the number of insertions made in the array or it might be a limit on the number of data comparisons. The output of a testable algorithm might be the state at some intermediate point in a computation.

We formally define a *testable algorithm* as a triple $T = (\mathcal{I}, \mathcal{O}, q)$, where $\mathcal{I}$ is the set of input instances, $\mathcal{O}$ is the set of outputs, and $q : \mathcal{I} \times \mathbb{N}_\infty \to \mathcal{O}$ is a function that takes an input and an execution parameter (which may be a natural number or infinity) to the outputs. Ordinarily $q(I, n)$ is evaluated by executing an ordinary algorithm, starting in a state derived from the input instance I, until the algorithm terminates or until some quantity reaches the limit value of n. The output is then some simple function of the state.

Consider, for instance, the following insertion sort algorithm.

```
1  for (int i = 1; i < n; i = i+1) {
2    int next = a[i];
3    int j = i;
4    while ((j > 0) && (a[j-1] > next)) {
5       a[j] = a[j-1];
6       j = j-1;
7    }
8    a[j] = next;
9  }
```

We can generate the following four testable algorithms:

—T_1: the execution parameter n limits the number of insertion steps, each input instance is an initial value for the array, and the output is the array's state when line 8 has been executed n times.

—T_2: the execution parameter n limits the number of insertion steps, each input instance is an initial value for the array and one of its positions, and the output is the value contained in the array at the specified position after line 8 has been executed n times.

—T_3: the execution parameter n limits the number of element comparisons, each input instance is an initial value for the array, and the output is the array's state after the expression `a[j-1] > next` has been evaluated $n + 1$ times.

—T_4: the execution parameter n limits the number of element comparisons, each input instance is an initial value for the array and one of its positions, and the output is the value contained in the array at the specified position after the expression `a[j-1] > next` has been evaluated $n + 1$ times.

Making use of the notion of a testable algorithm, we now introduce our categorization of multiple-choice quizzes on algorithms; for each kind of test, we will also describe how the quiz components can be automatically generated.

2.1 Input selection quiz

Given a testable algorithm $T = (\mathcal{I}, \mathcal{O}, q)$, an *input selection quiz* consists of a question $(I_1, I_2, \ldots, I_k, O, n)$, where $k > 1$, $I_j \in \mathcal{I}$ for each j such that $1 \leq j \leq k$, $O \in \mathcal{O}$, and $n \in \mathbb{N}_\infty$. The multiple-choice answers are the k values $1, 2, \ldots, k$. A choice j is correct if and only if $q(I_j, n) = O$.

In order to implement such a quiz, we can "randomly generate" an instance $I \in \mathcal{I}$, randomly generate n, compute $O = q(I, n)$, and "slightly perturb" I in order to obtain alternative instances which might appear reasonable for O. Finally, in order to determine the correct choices, we evaluate $q(I_j, n)$ for each instance I_j, and check whether the result is equal to O. An example of an input selection quiz is shown in the left part of Figure 1; in this case, the perturbation of the randomly generated sequence has been obtained by simply permuting its elements in order to obtain two other possible sequences.

2.2 Output selection quiz

Given a testable algorithm $T = (\mathcal{I}, \mathcal{O}, q)$, an *output selection quiz* consists of a question $(I, O_1, O_2, \ldots, O_k, n)$, where $k > 1$, $I \in \mathcal{I}$, $O_j \in \mathcal{O}$ for each j such that $1 \leq j \leq k$, and $n \in \mathbb{N}_\infty$. The multiple-choice answers are the k values $1, 2, \ldots, k$. A choice j is correct if and only if $q(I, n) = O_j$.

In order to implement such a quiz, we can "randomly generate" an instance $I \in \mathcal{I}$, randomly generate n, compute $O = q(I, n)$, and "slightly perturb" n in order to obtain alternative output values which might appear reasonable for I. Even in this case, it is easy to generate the multiple choices. Finally, in order to determine the correct choices, we evaluate, for each value n_j, $q(I, n_j)$ and check whether the result is equal to O. An example of an output selection quiz is shown in the right part of Figure 1; in this case, the perturbation of the execution parameter has been obtained by simply increasing and decreasing its value by one.

2.3 Algorithm selection quiz

Given k testable algorithms $T_j = (\mathcal{I}, \mathcal{O}, q_j)$ for $1 \leq j \leq k$, an *algorithm selection quiz* consists of a question (I, O, n), where $I \in \mathcal{I}$, $O \in \mathcal{O}$, and $n \in \mathbb{N}_\infty$. The multiple-choice answers are the k values $1, 2, \ldots, k$. A choice j is correct if and only if $q_j(I, n) = O$.

In order to implement such a quiz, we can "randomly generate" an instance $I \in \mathcal{I}$, "randomly generate" a value n, and compute $q_j(I, n)$ for each q_j. Finally, in order to determine the correct choices, we evaluate $q_j(I, n)$ for each q_j, and check whether the result is equal to O. An example of an output selection quiz is shown in the left part of Figure 2.

2.4 Parameter value selection quiz

Given a testable algorithm $T = (\mathcal{I}, \mathcal{O}, q)$, a *parameter value selection quiz* consists of a question $(I, O, n_1, n_2, \ldots, n_k)$, where $k > 1$, $I \in \mathcal{I}$, $O \in \mathcal{O}$, and $n_j \in \mathbb{N}$ for any j with $1 \leq j \leq k$. The multiple-choice answers consists of the k values $1, 2, \ldots, k$. A choice j is correct if and only if $q(I, n_j) = O$.

In order to implement such a quiz, we can "randomly generate" an instance $I \in \mathcal{I}$, "randomly generate" n, "slightly perturb" n in order to obtain the other $k - 1$ parameter values. Finally, in order to determine the correct choices, we evaluate $q(I, n_j)$ for each value n_j and check whether the result is equal to O. An example of a parameter selection quiz is shown in the right part of Figure 2; in this case, the perturbation of the execution parameter has been obtained by simply increasing and decreasing its value by one.

2.5 Output-algorithm matching quiz

Given k testable algorithms $T_j = (\mathcal{I}, \mathcal{O}, q_j)$ for $1 \leq j \leq k$, an *output-algorithm matching quiz* consists of a question $(I, O_1, O_2, \ldots, O_k, n)$, where $I \in \mathcal{I}$, $O_j \in \mathcal{O}$ for each j such that $1 \leq j \leq k$, and $n \in \mathbb{N}_\infty$. The multiple-choice answers are the k^2 pairs (j, h) with $1 \leq j, h \leq k$. A choice (j, h) is correct if and only if $q_j(I, n) = O_h$.

In order to implement such a quiz, we can "randomly generate" an instance $I \in \mathcal{I}$, "randomly generate" a value n, and "randomly generate" a permutation π of $\{1, 2, \ldots, k\}$; then let $O_j = q_{\pi(j)}(I, n)$ for each j with $1 \leq j \leq k$. The multiple choices can be shown by means of a matrix. Finally, in order to determine the correct choices, we evaluate whether $O_h = q_j(I, n)$ for each value pair (j, h) with $1 \leq j, h \leq k$. An example of an output-algorithm matching quiz is shown in Figure 3.

2.6 Features required

The above categorization required that the following features be added to the Teaching Machine/WebWriter++ environment in order to implement all the classes of quizzes. First, a random instance generator and an instance perturbation mechanism were needed. Second, an output comparator had to be made. Third, an execution parameter perturbation mechanism was needed.

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 programming animation tool written in Java, and WebWriter++, which is a JavaScript library for authoring interactive web pages for teaching and learning programming.

3.1 Visualization plugins

In particular, 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 an 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++ algorithm for sorting an array, 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.

(1) A JavaScript command within the web page invokes the execution of the Java or C++ code within the Teaching Machine.
(2) 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.
(3) 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 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 in 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

This expands in more detail the example given for the third type of test described in Section 1.1 (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:

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

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

Table I. Scripting commands

(2) The value range desired for random population of the array.
(3) The sorting algorithm to be used.
(4) The number of sorting passes (or, again a permissable range).

The code (or really code sets, since different pieces of code would be required for different topics) and the visualizations form a resource base for creating actual quizzes. The quizzes themselves are 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

(1) Load the appropriate code into the Teaching Machine.
(2) Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.
(3) Start up the Teaching Machine to run invisibly (so the student cannot inspect it).
(4) Specify the visualizer and the data whose pictures we want.
(5) Have the Teaching Machine stop after it has executed the requisite number of sorting steps.
(6) Recover the two snapshots (before and after) from the visualizer.

Nevertheless, it requires far more detailed control of the Teaching Machine than we has ever exercised before starting this project, for example the requirement to run the Teaching Machine invisibly and pass it parameters. Such control was done previously by simple scripting from WebWriter++ generated pages. Once the Teaching Machine applet was loaded, JavaScript calls could readily invoke Teaching Machine applet functions. QuizWriter++ simply extends this capability by adding new functionality both to the Teaching Machine and to the scripts, for example creating an autorun mode in the Teaching Machine (previously it had always been run manually, like a debugger), and allowing the passing of arguments from a script.

That alone is not enough, however. We found it was also convenient to control the Teaching Machine from within the running code, that is, to allow the example running to issue commands directly to the Teaching Machine (such as when to halt or when to drop a snapshot). In essence, 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 I shows the current set of scripting calls.

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

(1) Load the appropriate code into the Teaching Machine.
(2) Pass it some parameters, such as the size of the array, the selection of the bubble sort implementation and the number of sorting steps.
(3) Start up the Teaching Machine to run invisibly (so the student cannot inspect it).
(4) Specify the visualizer and the data whose pictures we want.
(5) Have the Teaching Machine stop after it has executed the requisite number of sorting steps and take a snapshot.
(6) Rerun the Teaching Machine on every other sorting algorithm specified.
(7) For each algorithm, have the visualizer compare the state of the array after the reference algorithm to the state of the array after the current sort.
(8) Recover the two snapshots (before and after) from the visualizer.
(9) Recover data specifying algorithms that produced equivalent sorts.

The scripting calls in Table I 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 [Naps et al.(2003b)]. As stated in the introduction, it tries to avoid some of the impediments listed in [Naps et al.(2003a)] 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, [Naps et al.(2003b)] and [Naps et al.(2003a)] 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 [Jarc et al(2000)] and [Naps et al.(2000)], where web-based tools are presented and evaluated, and [Rößling and Häußge(2004)], 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 1.1. Nevertheless, the existing capabilities already span the entire space of controls needed, in the sense that they require all the structural extensions to the Teaching Machine that are needed.

6.1 Research questions

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 research questions that are at the core to the whole endeavour of automated testing:

(1) 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?

(2) Even if we are successful in (1), 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?

(3) If we are succesful in (2), can we produce a set of quizzes that are reasonably equivalent? That is, would students taking different quizzes from each other 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 allow these other issues to be tackled in the future.

REFERENCES

K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

M. P. Bruce-Lockhart. WebWriter++: A small authoring aid for programming. In *Proc. Newfoundland Electrical and Computer Engineering Conference*, pages 109–112, 2001.

M. P. Bruce-Lockhart, P. Crescenzi, and T. S. Norvell. Integrating test generation functionality into the Teaching Machine environment. *Proc. 5rd Program Visualization Workshop*, to be published in *Electronic Notes in Computer Science,* 2009.

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

M. P. Bruce-Lockhart and T. S. Norvell. Developing mental models of computer programming interactively via the web. In *Frontiers in Education Conference*, pages 3–8, 2007.

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

M. L. Cooper. Algorithm visualization: The state of the field. Master thesis at Virginia Polytechnic Institute and State University, 2007.

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

D. J. Jarc, M. B. Feldman, and R. S. Heller. Assessing the benefits of interactive prediction using web-based algorithm animation courseware. *SIGCSE Bull.*, 32(1):377–381, 2000.

A. Korhonen, and L. Malmi. Taxonomy of visual algorithm simulation exercises. *Proc. 3rd Program Visualization Workshop*, pages 118–125, 2004.

M. Krebs, 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*, pages 281–285, 2005.

M. Laakso, 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*, 4:49–68, 2005.

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

T. L. Naps, 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. *SIGCSE Bull.*, 35(4):124–136, 2003.

T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ—an environment to actively engage students in web-based algorithm visualizations. *SIGCSE Bull.*, 32(1):109–113, 2000.

T. L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35(2):131–152, 2003.

G. Rößling and G. Häußge. Towards tool-independent interaction support. In *Proc. 3rd International Program Visualization Workshop*, pages 110–117, 2004.

E. Vihtonen 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*, pages 371–372, 2002.

September 10, 2008
December 22, 2008