ABSTRACT
Exercises with an automatic evaluation can be helpful for students, if they cover meaningful tasks and are sufficiently challenging. We have designed an exercise system that combines exercise tasks with automatic evaluation and integrated algorithm animation. The paper describes the current status and sample tasks our system can handle.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer & Information Science Education - Computer Science Education

General Terms
Management

Keywords
CS1, exercise, support, automatic evaluation, animation

1. INTRODUCTION
Algorithms and data structures play a central role in many areas of computing education, especially in the first few semesters. Students typically need to become familiar with a large number of different algorithms and data structures. To ensure that they have understood these materials, they may be asked to implement some algorithms or answer questions, often in the form of homework assignments.

While the type of “submission” for such problems can be in many different forms or media, we will consistently use the (simplifying) term “exercise” for all such problems given to students for the remainder of this paper. This lets us abstract from homework or programming assignments, voluntary tasks, tasks in textbooks, hand-written solutions, submission of source (or compiled) code in some programming language, and other forms of (potentially digital) content.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

In this paper, we will first define the type of exercises we are targeting. We will then examine the state of art in related exercise support systems. In Section 4, we present the design for our exercise support system, followed by a brief description of how the system works. Section 6 provides some illustrative examples of exercise tasks that our system can currently handle. We conclude the paper with a summary and areas of further research in Section 7.

2. TARGETTED EXERCISE TASKS
A typical exercise for sorting algorithms is to “manually execute” the given algorithm. While students should be able to do so, we think that there are more interesting—and at least partially more advanced—tasks that could be given. We provide a short list of such questions here, all selected for a given array and an specific implementation of Quicksort and the associated pivot choice strategy:

- What is the total number of swaps for this array?
- What will be the third pivot element?
- What will be the final position of the value initially at position 2?
- How often will the value 5 change its position during the execution of Quicksort?
- In which recursive call will 6 be the pivot element (please enter 0 for “never”, otherwise 1...n)?
- With which element will value 4 be swapped in its first swap (please enter 0 if 4 is never swapped)?
- What value should be used for -100 in the array so that this element does not change its position?
- Provide an input array of at least length 10 so that the second pivot element chosen will be at position 5.

We believe that all of these questions are valid and that a good student should be able to work out correct or almost correct answers for them. Some of the questions may require the student to either manually trace the algorithm until “the right point”, or to apply a deeper understanding of the algorithm to reach the correct answer. Of course, we usually hope for the latter, but also appreciate the possible learning gained from the former approach.

We are not interested in providing these questions as a multiple choice quiz, where students may easily “determine”
the correct answer by simply trying out all possible (pre-defined and visible) answers in any order. Additionally, we do not want to hard-code the correct answer, although this would make automated evaluation very easy. Each of the above questions would work equally well with any array that contains (at least) the values referred to in the question. Accordingly, we want to be able to provide a broad range of related exercises, perhaps even with almost identical questions, but where the input arrays are different. This allows students who have solved a given question correctly to try the same question on a different array, in order to validate if they have really understood the algorithm and can thus transfer their answer. At the same time, the system should avoid creating a huge workload for the student.

Based on the Engagement Taxonomy [9], we want students to be more actively involved than merely copying answers from mostly a passively viewed visualization. Thus, tracing the execution of Quicksort is out of scope for this paper: the student only has to open a visualization of Quicksort on the given parameter, “fast forward” to the proper positions, and write down the state of the array at these positions. Conversely, while the answer to some of the above questions can also be found in a visualization of the algorithm, extracting it (e.g., by going through it step by step and counting the number of swaps) may actually cause more effort than trying to manually determine the answer!

To raise the attractiveness of the system for educators, the amount of work needed to transfer one exercise set to another by changing concrete variable or parameter values should be as low as possible. Apart from editing the input parameters, the educator should not have to determine the (changed) correct answer, since this can result in much work for some of the above questions. Instead, the system shall be able to automatically determine the correct answer, so that the educator has to replace only the input parameters and modify their occurrence in the questions, where applicable.

Our desired system should thus be able to:

1. Support “interesting” questions that ask about details in the algorithm’s behavior.
2. Allow the student to specify input parameters which shall cause a certain behavior; for example, see the last example in the above list.
3. Provide automated feedback on whether the student’s answer was correct or not.
4. Support creating new exercises from existing ones by simply replacing the input parameters and/or the question text, without having to manually determine the correct (changed) answer.
5. Reduce the student’s manual workload by visualizing the behavior of the algorithm on request.
6. Integrate the visualization of the algorithm on both the given parameters and on student-provided parameters for reflection and further exploration.

3. RELATED WORK

Several approaches have been developed to support the grading of exercise tasks. For example, students can submit their work digitally in a learning platform, such as Moodle [3]. There are also initiatives to augment learning management systems to better address computing education requirements [13]. Depending on the type of tasks, the answer may be graded automatically or wait for a human grader.

Other systems visualize algorithms to help students understand how they behave. Such “algorithm/program visualization” systems have been very well documented [7, 8, 12]. Some systems already combine visualization with learning materials, often in so-called hypertextbooks [10, 14, 15] which directly include the visualizations inside the page or start the tool using Java WebStart once the user clicks on the associated link. Alternatively, the visualization system can be started within a learning management system [7, 15]. Visualization systems typically do not support questions or are restricted to multiple-choice questions, and cannot handle questions such as outlined in Section 2.

Some visualization systems can also visualize the algorithm based on user-provided input data, for example JHAVÊ [8] and Animal [11]. Other systems like Trakla2 [6], Matrix2 (which is also used in Trakla2) [5], and the recently introduced Jype [4], provide “algorithm simulation exercises”. Here, the student has to manually perform a given algorithm, typically by dragging elements to their new or target position or clicking on buttons to cause a certain function, e.g., a rotation in an AVL tree. While this can be very helpful for students, the focus of this approach is not suited for our intentions.

Since none of the established tools could support the operations that we wanted, we set out to design our own system, which we will describe in the next Section.

4. SYSTEM DESIGN

Based on the requirements described in the previous Sections, students shall be able to select an exercise in the system, answer the exercise questions and submit their answers. After submission, they shall receive an almost instantaneous feedback on whether the answer was correct. Additionally, the system shall prepare an animation of the algorithm and allow students to access this animation. The animation shall help students in seeing what actually happened and thus pinpoint where and what they may have misunderstood.

This system requires a number of different functionalities:

1. The system must be able to simulate or execute the selected algorithm.
2. It must be able to visualize the selected algorithm.
3. It must be able to evaluate the student’s answer.
4. Writing a new exercise should not require changing any existing (Java) code.

We decided to base the implementation of our system on the established Animal system, since this system already supports the ad-hoc generation of animation content based on user-provided data [11]. It therefore directly fulfills requirements 1 and 2. Generating a visualization based on user input does not require any special effort from the educator, assuming that a suitable generator already exists.

We wanted to ensure that the system is also usable by educators with limited or no Java programming experience. The determination whether the answer was correct should thus not occur in the generator or in the underlying algorithm,
as this would require programming and code modifications whenever a new exercise question was implemented.

Instead, we have decided to use checkpoints, similarly to the “interesting events” used in several algorithm visualization systems since the 1980s [2]. This means that at the places where something “interesting” happens, the educator inserts an API call similar to the one shown in Figure 3 to store all relevant attributes and local variables into the checkpoint. Since this API call always has the same essential form, doing so does not require much programming skill. Based on the checkpoint data, the system will use a sequence of operations to determine if the question was answered correctly. The student then receives appropriate feedback. The student can also click on a button to see the animation of the concrete execution.

The execution proceeds as follows:

1. The student selects an exercise file, written in XML.
2. The XML specification is deserialized into concrete exercise-related objects.
3. The exercise is displayed to the student together with the associated exercise tasks.
4. The student provides input data, usually parameter values or answers to questions.
5. The algorithm is run on the input specified by educator and/or student. While the algorithm is running, relevant data is collected for use in the next step.
6. A script determines if the student’s input led to the expected result. It retrieves data entries from the checkpoints based on the logic implemented in the script. More information on this script is given in the next section. Based on the result of this process, appropriate feedback is generated and shown to the student.

An example of the student view, together with a graded answer submission, is shown in Figure 1.

5. A BRIEF LOOK BEHIND THE SCENES

This Section provides an overview of the system for determining if a student answer was correct. We first look at the evaluation scripts and then examine how a given code needs to be modified to dump the relevant data.

5.1 Evaluation Script

An evaluation script is used to determine if the user’s answer was correct. The goal of designing this script notation was to make it easy to use for educators. The evaluation script uses JXPath [1], a W3C-standardized notation used for navigating and querying graphs. The “graph” to be navigated in our case is the data gathered from the checkpoints.

Scripts can use variables to store results of any type, such as int values or complete lists of events. Currently, only ten functions are available, as follows:

- **Animate(x)** processes a list of checkpoints or events x and ensures that the associated animation steps are indicated in the table of contents for this animation.
- **Check(cond)** checks if the condition passed in in JXPath notation is correct. If this is not the case, the execution of the script ends. The user can also provide an optional comment as a second parameter which is shown if the condition is evaluated to false.
- **Comment(s)** shows the comment s to the student.
- **CommentIf(cond, s)** shows the comment s if the condition cond (given in JXPath notation) evaluates to true.
- **Debug()** works like the **Dump()** command in debug mode, but is ignored otherwise.
- **Dump()** dumps the list of all variables and their values as a comment. If variable names are passed in, only the names and values of these variables are dumped.
- **Equals(a, b)** tests if both parameters are equal.
- **Max(a, b, ...)** returns the maximum of all parameters.
- **Ok(...)** assigns a result to the exercise and ends the execution. Without parameters, the query is regarded as successful. A JXPath condition can also be passed in to determine the result. If two parameters are passed in, the exercise will be regarded as successful if they are equal.
- **Retrieve(exp)** executes the JXPath expression exp. The result is typically a list of data or a single piece of information. If a second parameter is provided, the query uses this object as the source for retrieving data; otherwise, the complete pool of knowledge is used.

Figure 2 shows a script that checks if the user provided an array array in which the second pivot element chosen by Quicksort will be 5. Line 1 uses a JXPath expression to retrieve the second interesting event labelled “pivotChange”, providing direct access to all associated variables at this point of the algorithm’s execution. The “sequence number"
is incremented during the simulation of the algorithm whenever a new iteration of the outer loop or a new recursion starts. Thus, the query in line 1 refers to all *pivotChange* events with a sequence number of 2. The resulting checkpoint data is stored in the internal variable `s`. Line 2 retrieves the field “pivotValue” from the object stored in line 1. Thus, `piv` holds the value of the second pivot element.

1. `s = Retrieve("events/pivotChange[seqnr = 2]");`
2. `piv = Retrieve("/pivotValue", s);`
3. `Debug();`
4. `Check("$piv = 5", "Incorrect");`
5. `Ok();`

Figure 2: Example Evaluation Script for an Exercise

Line 3 provides debug data visible only for educators. Line 4 checks—using JXPath notation—whether the retrieved value `piv` is equal to 5, and if not, marks the solution as incorrect. Otherwise, line 5 marks the solution as correct.

After a short familiarization, the evaluation script allows even complex queries to be performed easily.

### 5.2 How to Modify Existing Generators

As mentioned in Section 4, the implementation of the visualizations is based on the ANIMAL system [11]. The visualizations will be generated on-demand by so-called “generators”. A generator is a Java class that implements a special interface and contains (among others) a generate method that is responsible for creating the visualization. Of course, the actual work may also be delegated to service methods.

A given generator uses the same variables as any other implementation of the algorithm to faithfully execute the algorithm. Thus, all values are present in the generator in the form of local variables or object attributes. By using the checkpointEvent method, the educator can generate a simple checkpoint, as shown in Figure 3.

1. `CheckpointUtils.checkpointEvent(this, "pivotChange", new Variable("pivotValue", pivot), new Variable("pivotIndex", r), new Variable("leftIndex", l), new Variable("rightIndex", r), new Variable("animstep", lang.getStep()));`

Figure 3: Dumping Relevant Data to a Checkpoint

### 6. EXAMPLE EXERCISE TASKS

Table 1 lists some of the tasks that we have implemented. All of these tasks work on data that the educator has specified, for example a concrete array of values to be sorted. The educator only has to copy a given exercise specification, say for CombSort, and enter a different base array to generate a new “exercise sheet” with the same tasks, but working on a different basic input. Nothing in the exercise logic has to be changed, so that most computing educators should be able to create a new exercise sheet from an existing one.

The scripts (as described in Section 5.1) to evaluate the correctness of the student’s answer typically have only 3-5 lines (not counting comments and debug information).

Figure 4 shows an example animation, started by clicking on the “Show me” button in Figure 1. The “table of contents” shows the interesting events in this animation, and draws attention to the swap events by prepending them with “>>”.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Task</th>
<th>User input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman-Ford</td>
<td>Give the weight of the shortest path from the source vertex A to vertex D.</td>
<td><code>int x</code></td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>Start at node S, give the distance from S to V, S to T, and S to X.</td>
<td><code>int sv, st, sx</code></td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>Give the sorted order.</td>
<td><code>char n1 - n7</code></td>
</tr>
<tr>
<td>Kruskal</td>
<td>Sort all edges in increasing order of length and give the fourth edge.</td>
<td><code>char from, to</code></td>
</tr>
<tr>
<td>Kruskal</td>
<td>Give your third chosen edge.</td>
<td><code>char from, to</code></td>
</tr>
</tbody>
</table>

### Table 1: Example Tasks in the Exercise System

#### Searching Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Task</th>
<th>User input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolated Search</td>
<td>How many iterations are needed to find the value 2?</td>
<td><code>int x</code></td>
</tr>
<tr>
<td>ComSort</td>
<td>Which values are swapped when the gap size is 2?</td>
<td><code>int a, b</code></td>
</tr>
<tr>
<td>GnomeSort</td>
<td>How often is the value 1 swapped?</td>
<td><code>int x</code></td>
</tr>
<tr>
<td>ShakerSort</td>
<td>How many “shakes” are required to sort the array?</td>
<td><code>int x</code></td>
</tr>
<tr>
<td>ShakerSort</td>
<td>On the first “up” shake (from left to right), which will be the first indices to be swapped?</td>
<td><code>int a, b</code></td>
</tr>
<tr>
<td>ShellSort</td>
<td>Provide an array which uses 4 pivots to be sorted.</td>
<td><code>int[] array</code></td>
</tr>
<tr>
<td>Selection Sort</td>
<td>How many swaps will be performed in the following array [3, 5, 0, 2, 4, 1]?</td>
<td><code>int swaps</code></td>
</tr>
</tbody>
</table>

#### Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Task</th>
<th>User input</th>
</tr>
</thead>
<tbody>
<tr>
<td>CombSort</td>
<td>Provide an array that will be sorted by three swap operations.</td>
<td><code>int[] x</code></td>
</tr>
<tr>
<td>Quicksort</td>
<td>Provide an array which uses 4 pivots to be sorted.</td>
<td><code>int[] array</code></td>
</tr>
<tr>
<td>ShakerSort</td>
<td>On the first “up” shake (from left to right), which will be the first indices to be swapped?</td>
<td><code>int a, b</code></td>
</tr>
<tr>
<td>ShellSort</td>
<td>Provide an array which uses 4 pivots to be sorted.</td>
<td><code>int[] array</code></td>
</tr>
<tr>
<td>Selection Sort</td>
<td>How many swaps will be performed in the following array [3, 5, 0, 2, 4, 1]?</td>
<td><code>int swaps</code></td>
</tr>
</tbody>
</table>

#### Graph Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Task</th>
<th>User input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellman-Ford</td>
<td>Give the weight of the shortest path from the source vertex A to vertex D.</td>
<td><code>int x</code></td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>Start at node S, give the distance from S to V, S to T, and S to X.</td>
<td><code>int sv, st, sx</code></td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>Give the sorted order.</td>
<td><code>char n1 - n7</code></td>
</tr>
<tr>
<td>Kruskal</td>
<td>Sort all edges in increasing order of length and give the fourth edge.</td>
<td><code>char from, to</code></td>
</tr>
<tr>
<td>Kruskal</td>
<td>Give your third chosen edge.</td>
<td><code>char from, to</code></td>
</tr>
</tbody>
</table>

### 7. SUMMARY AND CONCLUSIONS

In this paper, we have presented a system for automatically evaluating student-submitted exercise solutions. The system supports the creation of new exercise sheets based on existing exercise sheets by simply replacing the input parameter, e.g., the array to be sorted. This therefore does not require any programming skills. The correct solution does not have to be given, since the system can automatically determine this based on short evaluation scripts.
Currently, the system covers only selected algorithms and exercise sheets. We are working on further exercise sheets for other algorithms. Most exercises were implemented—regarding both questions and evaluation scripts—by undergraduates, illustrating that the system is easy to use.

We cannot yet provide a formal evaluation of the system: the first exercises were only begun in November 2010, and our targeted lecture is scheduled for the summer term. Anecdotal evidence from students who have experimented with the system and our own experience indicates that the system can be very helpful. The integration of an animation of the algorithm means that students are not forced to manually “trace” the algorithm, considerably reducing the workload even for tasks that require multiple executions of an algorithm with “trial and error” input variations. This allows us to ask questions that we would not normally expect students to answer correctly, as shown in Table 1.

8. REFERENCES