Informatics in Education, 2004, Vol. 3, No. 2, 267–288

Visual Algorithm Simulation Exercise System with Automatic Assessment: TRAKLA2

Lauri MALMI, Ville KARAVIRTA, Ari KORHONEN,
Jussi NIKANDER, Otto SEPPÄLÄ, Panu SILVASTI
Department of Computer Science and Engineering, Helsinki University of Technology
P.O.Box 5400, 02015 HUT, Finland
e-mail: {lma,vkaravir,archie,jtn,oseppala,psilvast}@cs.hut.fi

Received: August 2004

Abstract. Interaction and feedback are key factors supporting the learning process. Therefore many automatic assessment and feedback systems have been developed for computer science courses during the past decade. In this paper we present a new framework, TRAKLA2, for building interactive algorithm simulation exercises. Exercises constructed in TRAKLA2 are viewed as learning objects in which students manipulate conceptual visualizations of data structures in order to simulate the working of given algorithms. The framework supports randomized input values for the assignments, as well as automatic feedback and grading of students' simulation sequences. Moreover, it supports automatic generation of model solutions as algorithm animations and the logging of statistical data about the interaction process resulting as students solve exercises. The system has been used in two universities in Finland for several courses involving over 1000 students. Student response has been very positive.

Key words: algorithms, data structures, algorithm animation, algorithm simulation, automatic assessment, computer science education.

1. Introduction

Many automatic assessment systems[1] have been developed over the last decade to aid grading of exercises in large computer science courses. The main application context has been checking programming exercises (Benford *et al.*, 1993; Jackson and Usher, 1997; Luck and Joy, 1999; Saikkonen *et al.*, 2001; Vihtonen and Ageenko, 2002). Other applications include grading algorithm exercises (Bridgeman *et al.*, 2000; Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000) and analyzing object-oriented designs and flowcharts (Higgins *et al.*, 2002).

The main purpose of these systems has been to reduce the workload of human teachers. However, there are other very important issues involved, as well. First, learners have the freedom to submit exercises from any place and at any time, a feature of great value in distance learning, as well as traditional environments. Second, most systems allow resubmission of exercises after feedback. This option can promote learning considerably, since

[1] We deliberately omit automatic checking of multiple choice questions here.

it supports independent self study and allows students to learn from and correct errors in real time without waiting for feedback from the teacher (Malmi *et al.*, 2002). Third, automatic assessment allows personal variation of the exercises (Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000), a feature that is impractical in large courses, in which the teacher marks the exercises manually.

The interaction that results from automatic assessment and feedback with a resubmission option enhances student learning. Other forms of interaction can also be employed for promoting learning. First, the environment can be designed to support various forms of collaboration among learners and teachers, such as chat, email or discussion groups. Second, the environment can incorporate active learning modules in the form of interactive applets that visualize or animate important concepts. A good example of this is the Snapshots project by Ross and Grinder (2002). Such active learning applets typically include simple forms of interaction, such as set-at-a-time control of a visualization with arbitrary "undo" capability. They also allow students to change input parameters to the visualization system or to construct entirely new visualization models that can be animated. Finally, advanced versions of these applets provide feedback regarding the correctness of student-constructed models. Third, applets can also be designed that provide students with formative and summative feedback, both textually and visually, that aids them in exploring the possible states of the system at hand.

In this paper we concentrate on the second and third advanced forms of interaction in the context of algorithm exercises. We emphasize that in our exercises a learner actually *manipulates* data structures using advanced context-sensitive visual operations instead of viewing ready made visualizations. Our application is a web-based learning environment for teaching data structures and algorithms.

At the Helsinki University of Technology (HUT) we have been developing software for supporting the data structures and algorithms course since the early 1990's, when the first version of the TRAKLA system (Hyvönen and Malmi, 1993) was built. That system was the first to implement *computerized algorithm simulation exercises*. In such exercises, the student simulated the working of algorithms on a conceptual level by examining and manipulating diagrams of the data structures to which those algorithms applied. Such manipulation was originally performed manually with pen and paper. The system just checked the answers, that is, the final states of the simulations, which were submitted to the system as email messages. An important issue in such exercises was that instead of concentrating on the implementation details, this method concentrated on promoting *conceptual understanding* of the algorithms involved. Moreover, each exercise was *individualized* for each learner by randomizing the initial values of the algorithm. The answer consisted of a sequence of data structure states that had to be coded into a predefined format.

Email based submissions required textual formats, which were impractical and error-prone. When the World Wide Web was introduced in mid 1990's, it became obvious that algorithm visualization methods could be incorporated into the system. This promised to allow platform independent graphical representations of conceptual diagrams of data structures and network to allow students to interact with them. A new system, WWW-TRAKLA (Korhonen and Malmi, 2000) was then implemented. It was the first web-based

learning environment to include algorithm simulation exercises. WWW-TRAKLA coupled the algorithm simulation exercise concept with the best practise in the algorithm visualization. Student interactions were based on direct manipulation of the visual representation of the data structures (Stasko, 1991); students could also browse the generated sequence of states forwards and backwards (Boroni *et al.*, 1996). At the same time, another very important feature was incorporated into the system: immediate feedback and resubmission of solutions.

Graphical manipulation in algorithm exercises was a huge improvement over text-based editing and evaluation by students was encouraging. However, feedback on solutions was still returned to students by email in a textual form that was not very informative.

Moreover, implementing new exercises was time consuming on the part of the instructor and it required a lot of programming and testing. Furthermore, even though students created answers graphically in terms of algorithm simulation, and were allowed to run the exercises step by step backwards and forwards, the model answers generated by the system were solely in textual form. Actually, they typically included only the final state of the data structure in question.

Due to these limitations, WWW-TRAKLA did not realize the full power of algorithm simulation; a new supporting framework was needed for TRAKLA. This new framework, called Matrix, was developed by Korhonen and Malmi (Korhonen and Malmi, 2002). Matrix is a general purpose framework for building algorithm animations and simulations. The framework allows building applications in which the user for the first time has *full control of data structure manipulations through GUI interaction*, using a method which we call *visual algorithm simulation* (Korhonen, 2003). Compared with direct manipulation, visual algorithm simulation allows true interaction with the underlying data structures and on-line modification of them. Simulation operations are carried out by performing context-sensitive drag-and-drop operations, triggering menu commands, and pressing push buttons. Thus, the user can modify, for example, the contents and relations of nodes, activate operations on abstract data types, and use different visual representations for a single data structure.

In this paper, we present the TRAKLA2 framework for the straightforward creation and publishing of interactive exercises for data structures and algorithms. TRAKLA2 is based on Matrix and allows for the full power of Matrix in building different types of visual algorithm simulation exercises. TRAKLA2 supports automatic assessment and provides a well-defined schema for programming new exercises. Our experience shows that new exercises can be built with the same or less time and effort as with the old system. However, the newly created exercises are fully visual and include visual algorithm simulation functionality and visual model solutions. This is much better than the old text based system. Moreover, TRAKLA2 provides automatic logging of data generated by user interaction with the system and automatic submissions (Silvasti *et al.*, 2004; Malmi and Korhonen, 2004). Thus, gathering data from exercise sessions for research purposes is easy.

Some other systems also incorporate visual algorithm simulation in exercises, such as PILOT (Bridgeman *et al.*, 2000) and SALA (Faltin, 2002). However, no others include

automatic assessment in terms of grading and storing submitted answers. Thus, these systems are more suitable for formative rather than summative assessment. For example,

PILOT allows stepwise execution of graph algorithms, but provides feedback only on a single step at a time and does not log nor grade a learner's work. Moreover, PILOT only works with graph algorithms, whereas TRAKLA2 can be used for exercises on basic data structures, sorting, searching, hashing and graph algorithms. In addition, TRAKLA2 supports both summative and formative assessment.

In the following sections, we describe the TRAKLA2 framework, its architecture and the process of creating new exercises. We also present some observations on TRAKLA2 exercises that were used in data structures and algorithms courses at the Helsinki University of Technology with a total enrollment of over 1000 students in years 2003 and 2004. The system has also been used at the University of Turku in a course of over 100 students.

2. Overview of TRAKLA2

TRAKLA2 (Korhonen *et al.*, 2003b) is a framework for automatically assessing *visual algorithm simulation exercises* (Korhonen *et al.*, 2003a). The system provides a Java applet that can display a variety of algorithms and data structures. TRAKLA2 also distributes individually tailored tracing exercises to students and automatically evaluates answers to the exercises. In visual algorithm simulation exercises, a learner directly manipulates the visual representation of the underlying data structures to which the algorithm is applied. The learner manipulates these real data structures through GUI operations with the purpose of performing the same changes on the data structures that the real algorithm would do. The answer to an exercise is a sequence of discrete states of data structures resulting from application of the algorithm, and the task is to determine the correct operations that will cause the transitions between each two consecutive states.

Let us consider the exercise in Fig. 1. The learner has started to manipulate the visual representation of the Binary Heap data structure by invoking context-sensitive *drag-and-drop operations*. In the next step, for example, he or she can drag the key C from a *Stream of keys* into the left subtree of R in the binary heap. After that, the new key is sifted up via a swap with its parent until the parental dominance requirement is satisfied (the key at each node is smaller than or equal to the keys of its children). The swap operation is performed by dragging and dropping a key in the heap on top of another key. In addition, the exercise applet includes a push button for activating the Delete operation. The `Delete` button is applied in the second phase of the exercise to simulate the deleteMin operation. Of course, there are several correct ways to heapify the tree while inserting or deleting a key, thus the drag-and-drop operations can also be targeted to "empty keys", and the deleteMin operation does not have to start by removing the root node (but by swapping it with the last node, in which case the delete should be targeted to the key at the last node). Thus, the delete operation is performed by selecting the target node before pressing the corresponding push button.

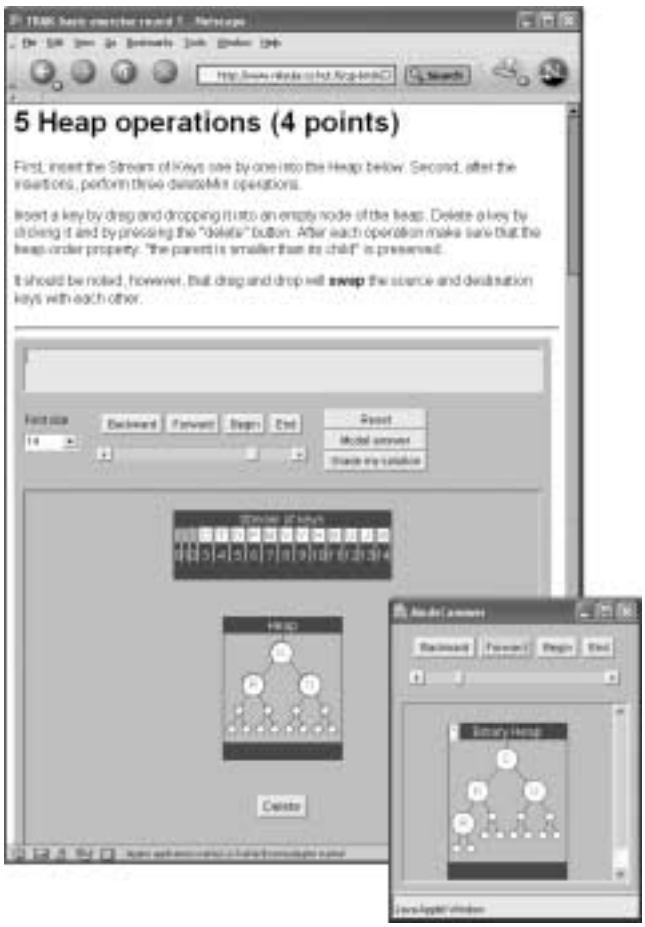

Fig. 1. TRAKLA2 applet page and the model solution window.

An exercise applet is initialized with proper *randomized input data*. The binary heap exercise, for example, is initialized with 15 alphabetic keys (Stream of keys), that do not contain duplicates. This means that the exercise can be initialized in more than 10^{19} different ways. The learner can *reset the exercise* by pressing the Reset button at any time. As a result the exercise is reinitialized with new random keys.

After attempting to solve the exercise, the learner can *review the answer* step by step using the `Backward` and `Forward` buttons. Moreover, the learner can *ask feedback* on his or her solution by pressing the `Grade` button in which case the learner's answer is checked and immediate feedback is delivered. The feedback reports the number of correct steps out of the total number of required steps in the exercise.

Fig. 2 shows an example of the feedback report. After the exercise is attempted, it is possible for the student to *submit the answer* to the course database using the Submit button. By default an answer to an exercise can be submitted unlimited times; however, a solution for a specific instance of exercise with certain input data can be submitted

Fig. 2. TRAKLA2 feedback window. A graded solution can be submitted to the course data base or one can cancel it and try to solve the exercise again with new input data values.

only once. In order to resubmit a solution to the exercise, the learner has to reset the exercise and start over with new randomized input data, that is, with a new instance of the exercise. Thus, it is not possible to grade the same solution and improve it arbitrarily before submitting it. However, TRAKLA2 also includes an option to create exercises, where improving the solution to the same exercise instance is allowed. In these cases, the number of allowed submissions is limited.

A highly important feature of TRAKLA2 is that a learner can *examine the model solution* of an exercise. It is represented as an algorithm animation so that the execution of the algorithm is visualized step by step. In Fig. 1, the model solution window is opened in the front. The states of the model solution can be browsed using the `Backward` and `Forward` buttons. For obvious reasons, after opening the model solution for given input data, a student cannot submit a solution until the exercise has been reset and resolved with new random data. Moreover, if in the exercise set-up the number of allowed submissions has been limited (the user continues operating on the same data), the model solution feature should be disabled before the submission deadline for the exercise is over.

TRAKLA2 was brought into production use in spring 2003 at the Helsinki University of Technology and at the University of Turku in 2004. The associated textual learning material is organized around the exercises, and each exercise is described in one page. Each TRAKLA2 exercise page (e.g., Fig. 1) consists of a description of the exercise, an interactive Java applet, and links to other pages that introduce the theory and examples of the algorithm in question. The current exercise set covers almost 30 assignments on basic data structures, sorting, searching, hashing, and graph algorithms. The complete list of exercises is included in the supplement of this article.

3. Frameworks

TRAKLA2 is based on the Matrix algorithm visualization, animation, and simulation framework (Korhonen and Malmi, 2002). Matrix provides the following reusable *fundamental data types* (FDT) to be used in the algorithm simulation exercises: arrays, linked lists, common trees, binary trees, and graphs. Fundamental data types are generic types which impose no semantic meaning on the data stored in them. Thus, they allow for primitive manipulations of structures, for example, changing keys and pointer values without

any constraints. However, the Matrix framework also provides an extensive library that includes ready-made implementations for many basic data types and abstract data types such as stacks and queues, search trees, and priority queues. Different implementations of these abstract data types, for example, as binary search trees, AVL trees, or radix search tries are called *conceptual data types* (CDT). They can be applied to provide more functionality to the simulation process.

The separation of concepts FDT and CDT clarifies both the design of the system itself and the range of possible simulation operations. Thus, the environment is not limited to supporting simple operations only (e.g., changing keys or manipulating pointers), but it also allows ways to implement and call more abstract operations, such as an insertion into an AVL tree or a deletion from a stack with single drag-and-drop operations. However, at the base all CDTs are implemented by reusing the FDTs. Thus, raising the level of abstraction requires no additional work what comes to the visualization, animation, or simulation. Moreover, for all reusable FDTs above (and thus also for the CDTs), there exist visual counterparts that enable the visualization and animation of a structure, as well as its simulation. We refer to these visualizations as *representations*.

TRAKLA2 heavily employs the different kinds of data structures and their representations provided by Matrix. When creating a new exercise for TRAKLA2, one can choose to (re)use any combination of the Matrix structures. For example, the exercise class (`ExerHeap`) in Fig. 1 uses an array FDT to hold the keys to be inserted into the heap, and the array implementation of the binary tree FDT to implement the actual binary heap CDT. The array implementation of the binary tree can thus be represented on the computer monitor as a tree or as an array. Thus, through reuse of these abstractions, the programmer does not have to implement any graphical objects to have a particular representation or GUI interaction, but he or she can concentrate entirely on implementing the essential parts of the exercise (e.g., the functionality of the binary heap).

The layout for the representation can be adjusted to better suit the exercise. For example, one can choose for the layout of the binary heap an array representation or a binary tree representation, as depicted in the Fig. 1. Both representations can be displayed simultaneously, allowing dual views of interaction with the data structure at the same time, if desired.

3.1. *Creating New Exercises*

As explained in Section 2, each TRAKLA2 exercise is placed on an exercise page in the web with the exercise description, links to related information, and an exercise applet. The applet is the same for all the exercises. Only a single Java class for each specific exercise distinquishes among exercises.

The framework provides much flexibility over the implementation of an exercise. To ensure that the original idea of the lecturer is conveyed into the implementation, the process of creating an exercise is usually begun by writing a detailed description. Based on this exercise manuscript, a programmer should be able to implement the exercise in Java using the features and components of the TRAKLA2 system. The resulting Java class defines the elements of the exercise by implementing several Java interfaces.

In general, each constructed exercise defines the following elements:

1) initialized data structures employed in the exercise (e.g., array, binary tree);
2) names for the visual representations (e.g., “Stream of keys”, “Heap”);
3) the layouts for GUI component placements and for each visual representation;
4) an algorithm for creating the model solution for a problem instance with the specified initial values (as well as a simple method to return the corresponding learner-made solution);
5) push buttons for the exercise, if required;
6) the allowed/disabled user interface operations for a representation (e.g., to determine whether it is allowed to drag and drop keys into the structure).

Each exercise is implemented as a Java class that minimally implements methods for items 1, 2, and 4. In the following we give a class definition for the example exercise discussed of Fig. 1.

`ExerHeap` is basically a class for a standard heap implementation that reuses and extends the Matrix library components (FDTs and CDTs). The overhead that comes from this approach is that instead of using the standard Java array type one must use the Matrix `VirtualArray` class to store the binary heap. The following example contains just the code needed to produce a *binary heap exercise*.

```
public class HeapInsertDelete extends AbstractSimulationExercise,
  ButtonExercise, ConfigureVisualType {
  private ExerHeap studentHeap ;
  private String randomInput ;
```

Each exercise has randomized input values. Although our example exercise could be initialized with *any* set of keys, an exercise may have special requirements for the input values (for example, to avoid too trivial solutions). The values are therefore tailored by a particular initialization algorithm. The framework supports easy input creation by providing reusable methods for creating different kinds of input data structures as in the example at hand. In this case, instructor’s manuscript of the *Binary Heap* (size $= 15$) exercise (described in Table 1) allowed no duplicate keys. The method was thus built to return a table (`new Table(randomInput)`) containing 15 different random keys to be inserted into the initially empty binary heap (`studentHeap`).

```
public FDT[] init() {
  randomInput = RandomKey.createNoDuplicateUppercaseRandomKey(
        new java.util.Random(seed), 15);
  studentHeap = new ExerHeap (15);
  return new FDT[] { new Table(randomInput), studentHeap};
}
```

The data structures returned by the initialization routine (as well as the model solution) are named by text strings returned by another method shown below. There also exists a method for delivering some additional information to the learner in form of text. In the heap exercise, it is left empty.

```
public String[] getStructureNames() {
  return new String[] { “Stream of keys”, “Heap” };
}

public String[] getModelAnswerNames() {
  return new String[] { “Binary Heap” };
}

public String getDescription() {
  return ”” ;
}
```

An implementation for the algorithm in question is essential for two reasons: to check the learner’s answer, and to produce the model solution as an algorithm animation. In this example, the algorithm uses the insertion and deletion routines that are already implemented in the `ExerHeap` class. These are just the standard implementations for manipulating a binary heap. The following `solve` method returns the model solution for this exercise by invoking the `insert` method for each key to be inserted and thereafter performing a `deleteMin` three times. Each method invocation is encapsulated between the annotated animator operations that define the boundaries of the animation steps, effectively setting the granularity of the animation. After the `solve` method is run the animator contains an animation sequence that can be traversed backward and forward. Moreover, in each animation step the binary heap can be visualized for the learner, if he/she has requested to see the model solution.

```
public FDT[] solve() {
  ExerHeap modelAnswer = new ExerHeap(15);
  Animator animator = Animator.getActiveAnimator();
  Table tbl = new Table(randomInput);

  // First we insert all keys to the heap
  for (int i = 0; i < tbl.size(); i++) {
   animator.startOperation();
   modelAnswer.insert(tbl.getObject(i));
   animator.endOperation();
  }
  // Then we perform 3 deleteMin/Max operations to the heap
  for (int i = 0; i < 3; i++) {
   animator.startOperation();
   modelAnswer.deleteMin();
   animator.endOperation();
  }
  return new FDT[] { modelAnswer };
}
```

In constrast to the old TRAKLA system, there is no need to implement a checking algorithm that evaluates solutions to an exercise. This is because TRAKLA2 evaluation is based on running the implemented algorithm and comparing the model answer step by step with the simulation sequence generated by the learner. Thus, TRAKLA2 only

requires the structures manipulated by the learner be returned when the learner requests grading[2].

```
public FDT[] getAnswer() {
  return new FDT[] { studentHeap };
}
```

Drag-and-drop operations may not be suitable for all kinds of data structure manipulations. For example, in the binary heap exercise the deleteMin operation is performed by selecting the root node and pressing the Delete button depicted in Fig. 1. The following two methods can be applied to gain such functionality.

```
public String[] buttonNames() {
  return new String[] { "Delete" };
}

public String[] buttonCommands() {
  return new String[]
         { "reflectSelectedVisualType(reflectEDT(deleteRoot))" };
}
```

The Matrix framework allows various user interface operations on the data structure representations. Therefore, it is important to define which operations are enabled and which are disabled for each structure in a TRAKLA2 exercise. For example, in Fig. 1, the task of the learner is to drag and drop keys *from* the array representing the stream of keys *into* the binary heap. Therefore, dragging keys from the table must be enabled, while dropping a key into the table must be disabled. The operations that can be enabled (or disabled) for each visual representation include: dragging the component, dropping the component, highlighting a component as the cursor is moved over it, and whether a popup menu can be opened upon the component.

```
public VisualTypeConf[] conf() {
  VisualTypeConf table = new VisualTypeConf();
  table.enable("matrix.visual.VisualKey",
       VisualTypeConf.HIGHLIGHT_OPERATION);
  table.enable("matrix.visual.VisualKey",
       VisualTypeConf.DRAG_OPERATION);
  VisualTypeConf tree = new VisualTypeConf();
  tree.enable("matrix.visual.VisualKey",
       VisualTypeConf.HIGHLIGHT_OPERATION);
  tree.enable("matrix.visual.VisualKey",
       VisualTypeConf.DROP_OPERATION);
  tree.enable("matrix.visual.VisualKey",
       VisualTypeConf.DRAG_OPERATION);
  tree.enable("matrix.visual.VisualKey",
       VisualTypeConf.POP_UP_MENU_OPERATION);
  tree.enable("matrix.visual.VisualLayeredTreeComponent",
       VisualTypeConf.HIGHLIGHT_OPERATION);
```

[2]Providing a checking algorithm is possible in cases where the validity of the learner's answer cannot be evaluated in the way just described. For example, in the Faulty Binary Search Tree exercise the given checking algorithm confirms that the student has managed to create an inconsistent state of the tree with the faulty tree operations.

```
    tree.enable("matrix.visual.VisualLayeredTreeComponent",
         VisualTypeConf.DROP_OPERATION);
    return new VisualTypeConf[] { table, tree };
  }
```

Finally, the TRAKLA2 exercise framework provides several functions that the exercise programmer does not need to know about. For example, it does not matter whether the example exercise is created only for trying things out or whether it is a compulsory exercise on which student performance is collected. In both cases the exercise class defined above would remain unchanged. An exercise can be *published* by adding it into a web page using a Java applet provided with the framework. These issues are discussed further in Section 4.

4. Learning Environment

All TRAKLA2 exercises are presented as applets that are merely user interfaces for algorithmic exercises. The applet must be published by embedding it into a web page, which contains the description of the assignment. In order to do this we have implemented a complete *learning environment* around the TRAKLA2 exercises. This is a dynamic web-based environment in which learners can solve exercises. The environment stores learners' exercise results, and it allows learners to keep track of their progression and to access additional material on the exercises.

4.1. *Architecture*

The general architecture of the learning environment is presented in Fig. 3. The environment consists of three independent modules: the *RMI-server*, the *WWW-server* and the *database*. Furthermore, the system writes several *log files*.

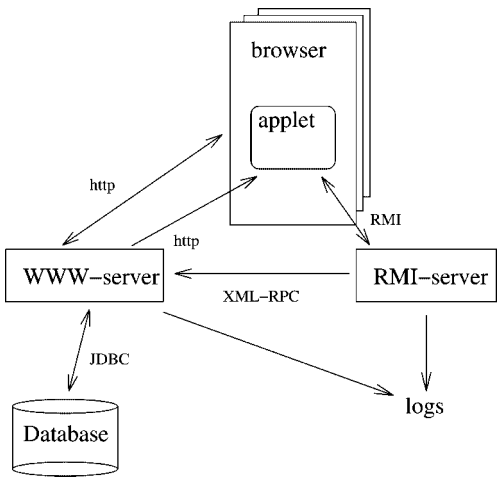

Fig. 3. Architecture of the TRAKLA2 learning environment.

The WWW-server is the heart of the system. It takes care of session handling, creation of dynamic web pages (including the exercise pages), initialization of the TRAKLA2 exercise applets, and storing the learners' results into the database. The WWW-server also handles all communication between different modules of the system.

The web environment is created using Tapestry (Apache Software Foundation, 2004), and Jetty (Mort Bay Consulting, 2004). Tapestry is a Java-based framework that allows for easy creation of dynamic web applications. The Tapestry engine runs over the Jetty web-server, which includes a servlet-engine as required by the Tapestry framework. The Jetty web-server is also Java-based, making the whole learning environment Java-based.

The Tapestry engine hides most of the unpleasant technical details required to run the learning environment. For example, it automatically handles the session control. Dynamic web pages are created from simple templates with the help of some Java code. The page templates are HTML pages including some Tapestry-specific tags that are expanded to more complex HTML definitions by the engine. All the Java code required for the creation of a single web page resides in a separate class file, which is linked to the HTML template by Tapestry. TRAKLA2 applets are initialized using one page template, which automatically includes all parameters required to run the exercise applet.

The RMI-server is a rather simple module that handles the communication between the TRAKLA2 exercise applets and the environment. The applet and the server communicate using Java's Remote Method Invocation (Sun Microsystems Inc, 2003) protocol. The TRAKLA2 applet connects to the RMI-server when it is initialized, and the server passes the applet a random seed for creating the randomized input data for the exercise. Moreover, the RMI-server also receives the learners' results. When a learner submits his or her answer, the RMI-server stores the solution as serialized Java objects. The points earned by the learner when a submission is graded are sent to the WWW-server using XML Remote Procedure Call (XML-RPC) (UserLand Software, 2003) protocol. Finally, the RMI-server also logs the data of the user interface operations the learner performs in the applet. This will be explained in more detail in Section 4.3.

All the data needed by the system is stored in a database that only the WWW-server accesses. The database contains user information, the results of learner's submissions, and information on the courses (exercises, deadlines, maximum points, grade limits, etc.). The database runs on a PostgreSQL server. The communication between the WWW-server and the database uses Java Database Connectivity (JDBC).

4.2. *Using the System*

When a learner starts to use the system, he/she is required to log in before accessing the exercises. Currently learners are assumed to be students, each having a unique *student identification number.* The identification of users also provides necessary data for session control.

After logging in the learner is directed to the main page, where he/she can view the active course or courses in which he/she is registered. If the learner is not registrated for any course, he/she is taken to the registration page. After registration the learner is directed to the main page.

On the main page (see Fig. 4) learners are shown the current status of his/her courses: points gained from each exercise, the total points gained from all exercises, the grade awarded with current points, and the state of each exercise. Each exercise has one of the three possible states. Exercises which the learner has not yet submitted are marked as "not started", the exercises the learner has submitted but has not received full points for are marked "started", and the exercises the learner has received full points for are marked "completed". The status of each exercise is shown by coloring the link to the exercise with a different color. Furthermore, each exercise is either open (the deadline has not passed yet) or closed (the deadline has passed). By default the learner can return closed exercises, but cannot receive any points for them.

From the main page a learner can select exercises. Each exercise is opened on its own page, which includes the exercise description, a link to available material, and the exercise applet itself. There are also navigation links to previous and next exercises as well as a link to return to the main page.

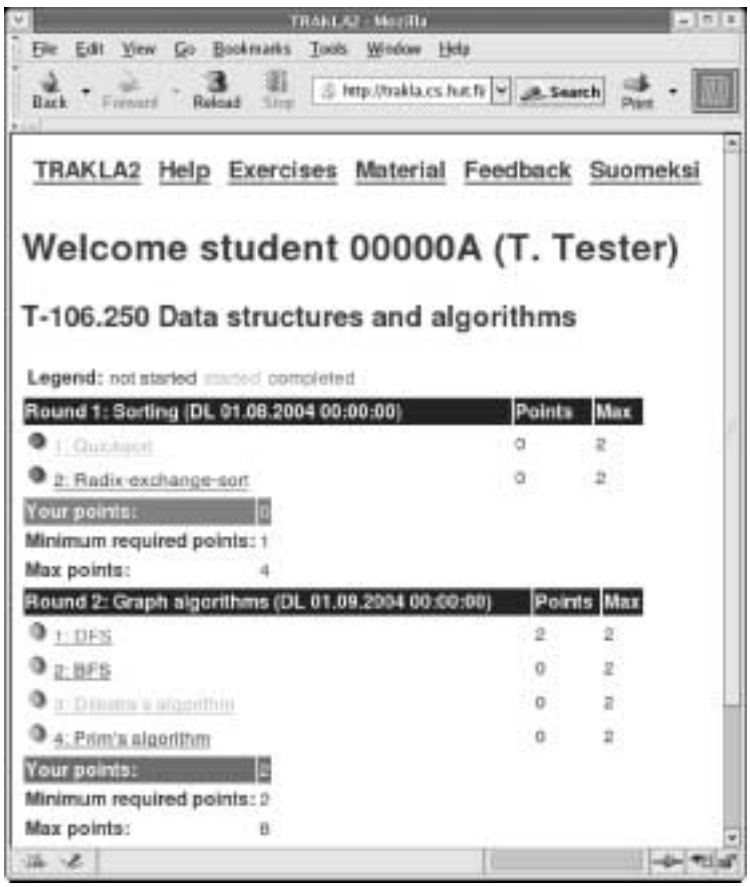

Fig. 4. Main page of the learning environment.

4.3. *Monitoring Students Performance*

In general, the graphical user interface of TRAKLA2 exercises is fairly simple. All basic operations – initialization of the exercise, opening the model answer, grading and submission – are the same in each exercise. The solutions are always defined in terms of algorithm simulation operations, i.e., by through manipulation of visual representations of data structures. The uniformity and simplicity of the user interface raises the possibility of collecting statistical data from the learner's interactions with the interface to better understand how learners interact with the system (Silvasti *et al.*, 2004).

The technical solution for collecting data is based on creating Java objects corresponding to the user interface operations. The objects are sent to a server through the Java RMI protocol. The server saves basic information of the log entries into a text file that is easy to analyze. A log entry is saved each time the applet is initialized, the model answer window is opened or closed, and the exercise is graded or reset. The entry contains a timestamp, course id, exercise id, student's identification id, and the name of the operation performed. A log entry is also saved each time the student is idle over 60 seconds, and when the idle time ends in a user interface operation. For each grade operation a snapshot dump of student's answer, which is a sequence of data structure states, is saved as a serialized Java object.

5. Experiences and Discussion

TRAKLA2 exercises were used for the first time in the basic data structures and algorithms courses at Helsinki University of Technology (HUT) in spring 2003. There were two versions of the course, one for CS majors and one for students of other engineering curricula. The system was used in parallel with the old TRAKLA system so that in total 14 TRAKLA2 exercises and 24 TRAKLA exercises were used in both courses. In 2004 only TRAKLA2 was used and the total number of exercises was 26. During these two years more than 1000 students used the system. In 2004, the University of Turku (UTU) also adopted TRAKLA2 for their data structure course with more than 100 students. In all of these courses TRAKLA2 exercises were a compulsory part of the course, and grading points achieved from the exercises had an effect on the final grade of the courses.

As a whole the system has worked well with surprisingly good results. In 2004 30% of the students at HUT achieved the maximum number of points for all of the 26 exercises, and 55% achieved at least 90% of the maximum. Only 15% of the students failed to get the required minimum of 50% of the points; in practice these were students who dropped the whole course early. At UTU the results were even better, even though they had never used the TRAKLA system before.

Students' opinions of the system were determined through a web-based survey at the end of the HUT course in 2003. 364 students answered. Their attitude towards the system was very positive: 94% of the students thought that the system was very easy to use. In addition, they considered the system a good learning aid: only 1% thought

that the feedback provided by TRAKLA2 was not useful, and only 3% did not get any help from the model solutions. Moreover, 96% of the students thought that the visual representations of the data structures were fairly logical or very logical. Finally, 80% gave an overall grade of 4 or 5 to the system in scale 0–5, where 5 was the best grade. The feedback questionnaire for the UTU course in 2004 had different questions, but the message was totally in line with the results at HUT. Students liked the system very much and considered it useful for learning.

There were, however, some technical problems. First, the TRAKLA2 applet requires a Java 2 compatible browser, and it appeared that in 2003 many popular browsers did not support Java 2 well enough. The problem was reduced in 2004. However, even though all HUT students were not able to get TRAKLA2 working on their home computers, they were able to use the system in the computer classrooms at HUT, where TRAKLA2 worked well. Second, the communication between the TRAKLA2 applet and the server did not work if there was a firewall blocking traffic on certain TCP ports at the client side. This problem appeared only in cases where the student was trying to use TRAKLA2 at his or her place of work. The security policy of a company firewall is usually stricter than, for example, at universities, libraries, and homes.

5.1. *Data Logging*

The data logging feature that is invisible to the exercise programmer is also very promising. We have gathered detailed information about the use of the applet and the model solutions, including how much time a student spends on an exercise. This can be used to test or verify hypotheses about the difficultiness of exercises, and to find out which phases in the algorithms students find the most difficult. This allows us to direct the instruction better to the actual needs. In this paper we do not present detailed results of our observations, but only point out with some sample cases what kind of information is possible to get through the logging facility.

The number of performed GUI operations (i.e., resets, gradings, submissions, model solution calls) provides information about the relative difficulty of the exercises. For example, Fig. 5 highlights the differences among average number of performed operations for some of the exercises in 2004. The error bars denote the 95% confidence range. The six exercises correspond to one round of exercises in the course. Usually the students also did their exercises in this particular order. The first two exercises (how binary and interpolation search algorithms proceed in a sorted array) were very similar in their look and feel, as well as the following three exercises (the order in which the nodes are traversed in a binary tree using different traversal algorithms). The last exercise dealt with preorder traversal in a binary tree but required showing the contents of the auxiliary stack, which complicated the exercise considerably.

At the first glance, the resuls may seem suprising. For example, binary search seems more difficult than interpolation search and there are big differences between the traversal algorithms. Therefore care must be taken to interpret the results. The largest differences appear with the number of initializations. This is, however, strongly connected to the

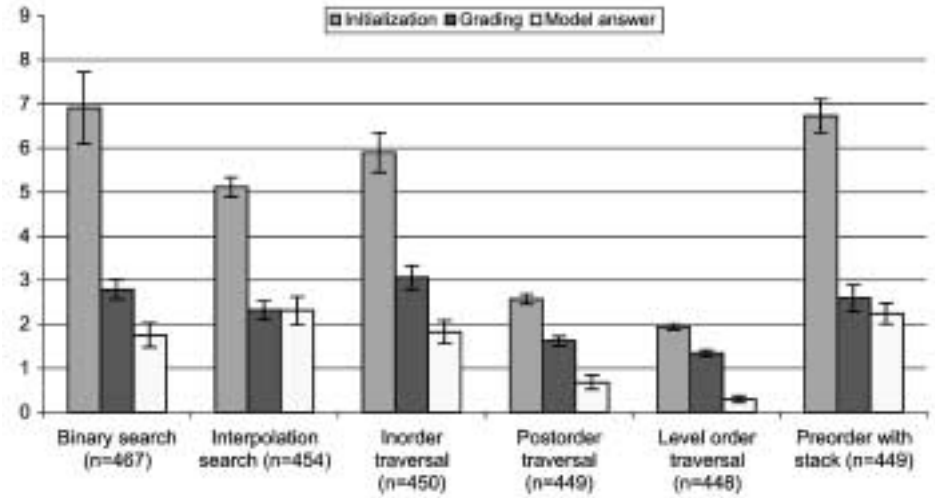

Fig. 5. Average operation numbers in different exercises. The sample sizes and error bars are included in the chart.

look and feel of the exercise. Students take their time to familiarize, how the exercises should be technically solved, i.e., which operations are needed and how they are carried out. Therefore they are likely to Reset a new type of exercise a few times before actually solving it. This can be clearly seen in the notable difference between the average numbers of initializations and gradings for the first exercise in each set of new types of exercises (i.e., exercises binary search, in order traversal and preorder traversal with stack). When the user interface of the exercise is better known, students concentrate readily on solving the exercise.

The relative difficulty of the exercises is better observed from the number of model solution activations. For example, for interpolation search model solutions are requested more ofhen than with binary search. For the traversal algorithms, when the students have grasped the general principle involved, the average number of gradings, as well as the correspoding need for viewing the model solution decreases. In other words, if you get the full points in the first trial, there is no point in viewing the model solution.

We did a much more elaborated survey (Torvinen, 2003) of the data on exercises in 2003 to better understand the results. We found out that the following information was consistently in line: how many times students had graded the solution, maximum points of the exercises they had received, the total time (excluding idle time) they had used for the exercises, and how many idle time stamps were recorded for the exercise. Thus, we could clearly see that insertions into red-black trees and AVL trees were the most difficult exercises among search trees (most gradings, least maximum points, most total time, and idle time used). Some more detailed results can be found in (Malmi and Korhonen, 2004).

The model solution is an animation that consists of a discrete sequence of visual snapshots of data structure states. The learner can browse the animation back and forth using the backward and forward buttons as depicted in Fig. 1. Fig. 6 shows the number of visits

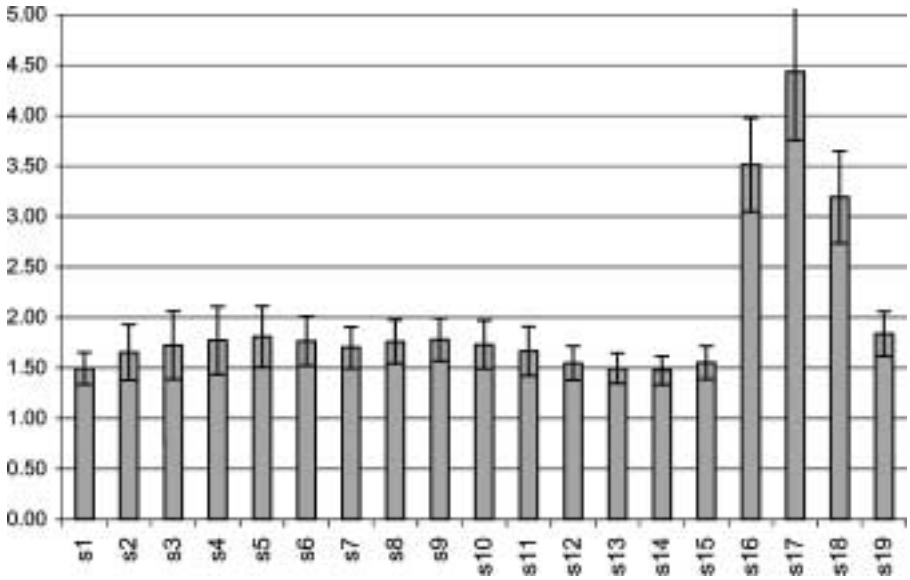

Fig. 6. The counts of students' visits in each state of the model solution animation of the heap insertion algorithm exercise. The learner has to perform 15 insertions and 3 deleteMin-operations.

in each state of the model solution in the binary heap exercise. The learner is to insert 15 keys into an initially empty binary heap and perform 3 deleteMin operations after that. The figure confirms the assumption that the deleteMin-operation is more difficult to understand than the insert operation. This is seen in that the steps among delete operations at the end of the model answer are viewed more often than the other states. The sample is taken from 73 students that looked at the model solution an average of 1.9 times.

6. Conclusion

As a whole, we argue that TRAKLA works well. Student response has been positive. Moreover, the implementation work on the exercises using the new framework has been quite straightforward. Although the exercises include visual manipulation of complex data structures, allow for automatic assessment, and provide model solutions as interactive algorithm animations, the work required to implement a new exercise was typically 2–3 workdays for a competent Java programmer if the exercise was initially clearly defined. A very important issue here is that the exercise programmer needs to have no deep knowledge about the Matrix framework. Knowing the basic concepts and interfaces is enough.

Actually the most challenging part has been designing the exercise specification manuscript: what kind of operations should be provided for the student in each case, which data structures – especially auxiliary data structures – should be shown on the screen, and what operations can be invoked for each data structure. For example, designing the quicksort exercise with a recursion stack was quite tricky. Several versions of the

exercise were implemented before we were satisfied with the overall user interface for the exercise.

The data logging facility has already proven its worth. We now have a tool which provides excellent opportunities to research students' learning processes, as we can monitor their work in many ways. This research is still in its formative stage. Finally, we note that such data logging might cause ethical problems if results on individuals were published. This should never be done. Such research results should be purely statistical and anonymous.

Acknowledgements

We thank Pekka Mård, Harri Salonen, Kimmo Stålnacke and Petri Tenhunen for designing and implementing major parts of the TRAKLA2 framework and the exercises. We also thank Rocky Ross for many valuable comments on the language of this paper.

References

Apache Software Foundation (2004). *Tapestry – Java Framework for Creating Web Applications*. `http://jakarta.apache.org/tapestry/`

Benford, S., E. Burke, E. Foxley, N. Gutteridge and A.M. Zin (1993). Ceilidh: A course administration and marking system. In *Proceedings of the 1st International Conference of Computer Based Learning*. Vienna, Austria.

Boroni, C.M., T.J. Eneboe, F.W. Goosey, J.A. Ross and R.J. Ross (1996). Dancing with Dynalab. In *27th SIGCSE Technical Symposium on Computer Science Education*. ACM, pp. 135–139.

Bridgeman, S., M.T. Goodrich, S.G. Kobourov and R. Tamassia (2000). PILOT: An interactive tool for learning and grading. In *The Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*. ACM, pp. 139–143.

Faltin, N. (2002). Structure and constraints in interactive exploratory algorithm learning. In S. Diehl (Ed.), *Software Visualization*: *International Seminar*. Springer, Dagstuhl, Germany, pp. 213–226.

Higgins, C., P. Symeonidis and A. Tsintsifas (2002). The marking system for CourseMaster. In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*. ACM, pp. 46–50.

Hyvönen, J., and L. Malmi (1993). TRAKLA – a system for teaching algorithms using email and a graphical editor. In *Proceedings of HYPERMEDIA in Vaasa*. pp. 141–147.

Jackson, D., and M. Usher (1997). Grading student programs using ASSYST. In *Proceedings of 28th ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, pp. 335–339.

Korhonen, A. (2003). *Visual Algorithm Simulation*. Doctoral thesis. Helsinki University of Technology, Laboratory of Information Processing Science, Report TKO-A40.

Korhonen, A., and L. Malmi (2000). Algorithm simulation with automatic assessment. In *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*. ACM, pp. 160–163.

Korhonen, A., and L. Malmi (2002). Matrix – Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*. ACM, pp. 109–114.

Korhonen, A., L. Malmi and P. Silvasti (2003a). TRAKLA2: a framework for automatically assessed visual simulation exercises. In *Proceedings of the Third Finnish/Baltic Sea Conference on Computer Science Education*. pp. 48–56.

Korhonen, A., L. Malmi, P. Silvasti, J. Nikander, P. Tenhunen, P. Mård, H. Salonen and V. Karavirta (2003b). *TRAKLA2*. URL: `http://www.cs.hut./Research/TRAKLA2/`

Luck, M., and M. Joy (1999). A secure on-line submission system. *Software – Practice and Experience*, **29**(8), 721–740.

Malmi, L., and A. Korhonen (2004). Automatic feedback and resubmissions as learning aid. In *Proceedings of 4th IEEE International Conference on Advanced Learning Technologies, ICALT'2004*. IEEE, pp. 186–190.

Malmi, L., A. Korhonen and R. Saikkonen (2002). Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of the 7th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'02*. ACM, pp. 55–59.

Mort Bay Consulting (2004). *Jetty – Web Server & Servlet Container.* `http://www.mortbay.org/jetty/`

Ross, R.J., and M.T. Grinder (2002). Hypertextbooks: animated, active learning, comprehensive teaching and learning resource for the web. In S. Diehl (Ed.), *Software Visualization*: *International Seminar*. Springer, Dagstuhl, Germany, pp. 269–283.

Saikkonen, R., L. Malmi and A. Korhonen (2001). Fully automatic assessment of programming exercises. In *Proceedings of the 6th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, ITiCSE'01*. ACM, pp. 133–136.

Silvasti, P., L. Malmi and P. Torvinen (2004). Collecting statistical data of the usage of a web-based educational software. In *Proceedings of the IASTED International Conference on Web-Based Education*. IASTED, pp. 107–110.

Stasko, J.T. (1991). Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of Conference on Human Factors and Computing Systems*. ACM, USA, pp. 307–314.

Sun Microsystems Inc. (2003). *Java Remote Method Invocation specification*. Available at: `http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html)`

Torvinen, P. (2003). Tilastollinen analyysi algoritmisten harjoitustehtäväsovelmien käytöstä (in Finnish, Statistical analysis of usage of algorithmic exercise applets). Helsinki University of Technology, Finland.

UserLand Software (2003). *XML-RPC specification*. Available at: `http://www.xmlrpc.com/spec`

Vihtonen, E., and E. Ageenko (2002). Viope-computer supported environment for learning programming languages. In *The Proceedings of Int. Symposium on Technologies of Information and Communication in Education for Engineering and Industry* (*TICE2002*). Lyon, France, pp. 371–372.

Supplement: TRAKLA2 Exercises

Table 1

The visual algorithm simulation exercises in TRAKLA2 system. The column name describes the topic and the description characterizes the exercise. The roman numbers (i–iv) indicate the separate exercises and the number of sub-topics.

Name	Description
Insertion into (i) Binary search tree and (ii) Digital search tree	The learner is to insert random keys into an initially empty search tree by dragging and dropping the keys into the correct positions.
Binary search tree deletion	The learner is to remove 4 keys from a binary search tree.
Faulty Binary Search Tree	The learner is to show how to bring the following binary search tree in an inconsistent state: duplicates are allowed and inserted into the left branch of an equal key, but the deletion of a non-leaf node relabels the node as its successor.
AVL tree (i) insertion, (ii) single rotation, and (iii) double rotation	The learner is to (i) insert 13 random keys into an initially empty AVL-tree. The tree (i–iii) has to be balanced by rotations. The rotation exercises (ii–iii) require pointer manipulation, while the insertion exercise (i) provides push buttons to perform the proper rotation at the selected node.
Red-black-tree insertion	The learner is to insert 10 random keys into an initially empty Red-Black-tree. The color of the nodes must be updated and the tree must be balanced by rotations.
BuildHeap algorithm	The learner is to simulate the linear time buildheap algorithm on 15 random keys.
Binary heap insertion and delete min	The learner is to a) insert 15 random keys into a binary heap and b) perform three deleteMin operations while preserving the heap order property (see Fig. 1).
Sequential search: (i) Binary search, and (ii) Interpolation search	The task is to show which keys the algorithm visits in the given array of 30 keys by indicating the corresponding indices.
Tree traversal algorithms: (i) preorder, (ii) inorder, (iii) postorder, and (iv) level order	The learner is to show which keys in a tree the algorithm visits by indicating the keys in the required order.
Preorder tree traversal with stack	The learner is to simulate how the stack grows and shrinks during the execution of the preorder algorithm on a given binary tree.
Fundamental Graph algorithms: (i) Depth First Search, and (ii) Breadth First Search	The learner is to visit the nodes in the given graph in DFS, and BFS order.
Minimum spanning tree algorithms: Prim's algorithm	The learner is to add the edges into the minimum spanning tree in the order that Prim's algorithm would do.
Shortest path algorithms: Dijkstra's algorithm	The learner is to add the edges to the shortest paths tree in the order that Dijkstra's algorithm would do.
Open addressing methods for hash tables: (i) linear probing, (ii) quadratic probing, and (iii) double hashing	The learner is to hash a set of keys (10–17) into the hash table of size 19.
Sorting algorithms: (i) Quicksort, and (ii) Radix Exchange sort	The learner is to sort the target array using the given algorithm.

L. Malmi is a professor of computer science in Helsinki University of Technology (HUT). He received his Doctor of Technology diploma in HUT in 1997. His main research area is computer science education including software visualization, automatic assessment, new educational methods, and evaluating how they improve learning.

V. Karavirta is a MSc student at Helsinki University of Technology. He has worked as a research assistant in the Matrix project for over two years. His research interests are in data structure and algorithm visualization and web-based learning environments.

A. Korhonen is a researcher at Helsinki University of Technology (HUT). He received his DSc (computer science) in 2003 at HUT. His research includes data structures and algorithms in software visualization, various applications of computer aided learning environments, and automatic assessment in computer science education.

J. Nikander is a MSc student at Helsinki University of Technology. He has worked as a research assistant in the Matrix project for three years, and has been assisting on the introductory data strctures and algorithms course during this time. His research interests are in data structures and algorithm visualization, and he is currently working with his master's thesis on the administrator's interface to the TRAKLA2 environment.

O. Seppälä is a researcher and a DSc student at Helsinki University of Technology. His research interest lies with the use of automatic program visualization and debugging tools in CS education.

P. Silvasti is a DSc student at Helsinki University of Techonology (HUT). He received his MSc (computer science) in 2003. His research interests are in educational technology for computer science education.

Algoritmavimo užduočių automatinio vertinimo simuliacinė sistema TRAKLA2

Lauri MALMI, Ville KARAVIRTA, Ari KORHONEN, Jussi NIKANDER,
Otto SEPPÄLÄ, Panu SILVASTI

Interaktyvumas bei grįžtamasis ryšys yra esminiai veiksniai, grindžiantys mokymosi procesą. Per pastarąjį dešimtmetį buvo sukurta nemaža automatinio vertinimo ir grįžtamojo ryšio sistemų, skirtų informatikai ar informacinių technologijų dalykams mokyti. Šiame straipsnyje supažindinama su nauja sistema – TRAKLA2, skirta interaktyviam algoritmų simuliavimui ir interaktyvioms užduotims kurti. Užduotys, sukurtos naudojantis TRAKLA2, pateikiamos kaip mokymosi objektai, kurie suteikia studentams duomenų struktūrų koncepcinę vizualizaciją ir leidžia imituoti pateiktų algoritmų funkcionavimą. Sukurtoji sistema turi automatinį grįžtamąjį ryšį, be to pripažįsta atsitiktines įvedimo reikšmes, pateikiamas vertinant, bei pati vertina studentų sukurtų algoritmų simuliacinius fragmentus. Sistema turi ir automatinį sprendimų generavimą, pateikiamą kaip algoritmų animaciją, taip pat registruoja statistinius duomenis, susijusius su interaktyvumu. Sistema išbandyta dviejuose Suomijos universitetuose su daugiau nei 1000 studentų. Studentų atsiliepimai buvo itin palankūs.