Taxonomy of Visual Algorithm Simulation Exercises

Ari Korhonen and Lauri Malmi
Helsinki University of Technology
Department of Computer Science and Engineering
Finland

`archie@cs.hut.fi, lma@cs.hut.fi`

Abstract

This paper presents a taxonomy for algorithm simulation exercises that allow to build learning environments that not only portray a variety of algorithms and data structures, but also distribute tracing exercises to the student and then automatically evaluates his/her answer to the exercises. The taxonomy systematically classifies the exercises into 8 separate categories that have 8 subcategories each. Each category is characterized and demonstrated by describing an example exercise that falls in the category.

The taxonomy provides a thinking tool to systematically diversify the set of possible simulation exercises. Thus, the taxonomy promotes new perspectives to come up with novel exercises of completely new genre. Moreover, we demonstrate a fully working web based learning environment that already includes implementations for such exercises.

1 Introduction

Today, algorithm animation is primarily utilized for both supporting teaching in the lectures and for studying in open and closed labs. A variety of systems are available for us on the web. However, from the pedagogical point of view, many of them lack the potential to give automatic formative or summative feedback on a student's performance, which is an essential factor in the learning process. Fortunately, the unambiguity of algorithms and data structures allows one to set up automatically assessed exercises and compare the student's solution to the correct model solution. This gives an opportunity to produce systems that not only portray a variety of algorithms and data structures, but also distribute tracing exercises to the student and then evaluate his/her answer to the exercises. One possible method for building such systems is *visual algorithm simulation* (Korhonen, 2003), which allows to practise, for example, such core CS topics as sorting algorithms, search trees, priority queues, and graph algorithms on a conceptual level without writing any code. When automatic evaluation of students' submitted work is included into the system, we refer to this as *automatic assessment and feedback of algorithm simulation exercises.*

Very few systems fully support algorithm simulation exercises with the automatic assessment and feedback capability. However, TRAKLA (Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000), and its follower TRAKLA2 (Korhonen et al., 2003) both do, and they form the basis of our discussion. Some other systems support algorithm simulation exercises, as well, but only within a limited scope. PILOT (Bridgeman et al., 2000) is targeted to tracing exercises, but covers only graph algorithms and provides only formative feedback. On the other hand, "stop-and-think" questions requiring an immediate response from the learner, such as introduced in JHAVÉ (Naps et al., 2000), can be interpreted to be algorithm simulation exercises, too. The learner is supposed to understand the algorithm either by mentally executing the algorithm or by doing such a simulation with paper and pen. Thus, the system illustrates exercises where the learner is asked to manually trace an algorithm on a small data set. However, the system does not give feedback on the correctness of such a trace, but merely asks questions that should confirm that the learner has understood the concept. Finally, if we generalize the concept algorithm to be any well-defined procedure to solve computational problems, we can also bring in tools such as the problets introduced by Krishna and Kumar (2001). They illustrate problem generators on the topic of precedence and associativity of operators in which the learner is to evaluate expressions by solving sub-expressions in correct order. The task is to solve exercises by following the predefined rules of operator precedence and associativity (that could also be expressed as an algorithm).

In this paper, we introduce a systematical classification for algorithm simulation exercises. Our aim is to illustrate the full range of exercises this method supports, and thus better discover their potential in supporting learning. The taxonomical evaluation divides the exercises into separate categories that are described by characterizing each category first. In addition, each category is demonstrated by describing an example exercise that falls in the category. We have a fully working web based learning environment that includes implementations for many of the exercises. Thus, you can try out the exercises in live action[1].

In Section 2, we briefly describe how to apply algorithm animation and simulation together to construct a *learning environment* with exercises for data structures and algorithms. Section 3 introduces the taxonomy of algorithm simulation exercises, and in Section 4 we present sample exercises in our learning environments TRAKLA and TRAKLA2. Finally, in Section 5, we discuss some aspects how this taxonomical approach could be applied in practice.

2 Algorithm simulation and animation in a learning environment

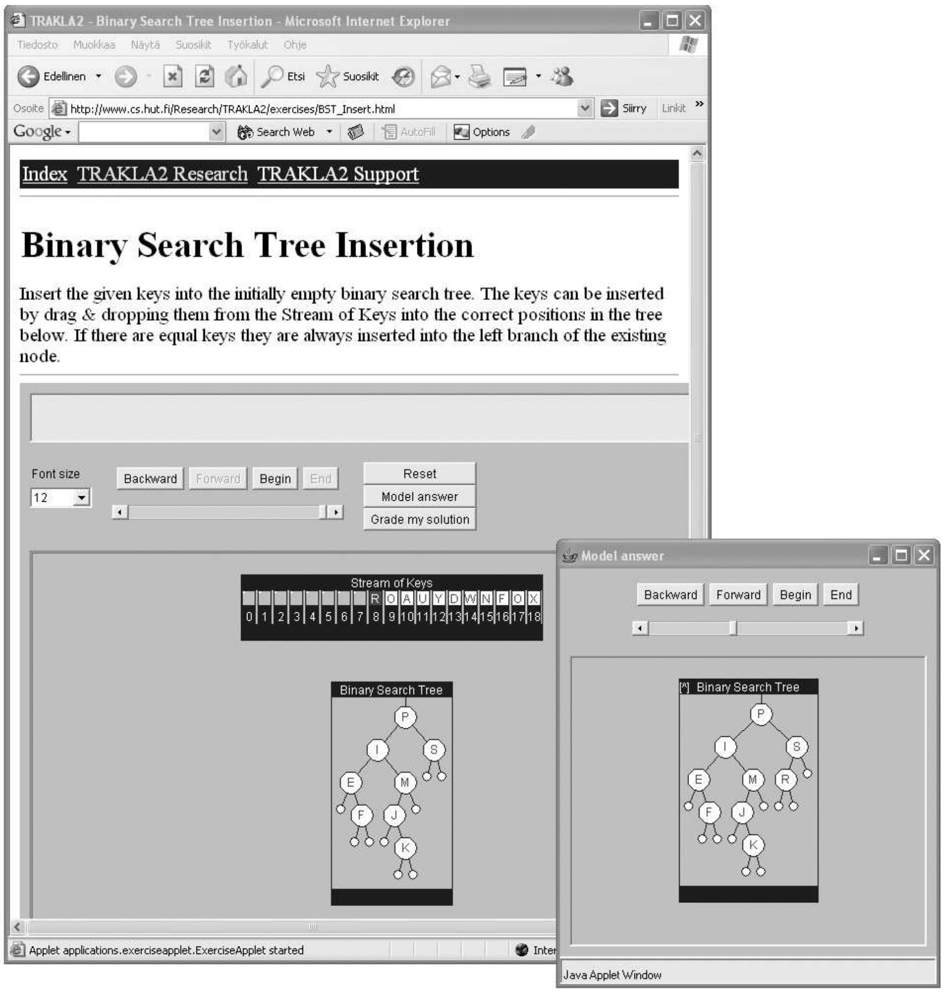
We define a learning environment as a system that is capable of meaningful interaction with the user with respect to the topic of the class to which this environment is attached. We refer to the users of such an environment as learners. In the context of data structures and algorithms, our vision is that a good learning environment should provide the learner a selection of interactive learning objects to view algorithms working (animation), interact with the animations, *i.e.*, control or simulate the algorithms, get feedback on the simulation in order to test one's knowledge on its working, and explore the general behavior of the algorithm through simulation with smaller or larger data sets. Moreover, the learner should be able to operate on the algorithm both on a conceptual level and on the implementation level to fully grasp its working. Implementing such a vision is a major task, and requires typically several different visualization tools. In this paper, we consider only working on the conceptual level, and the visualization of algorithm code execution is out of scope of this paper. First, we briefly define the concepts *algorithm simulation* and *automatic algorithm animation*, since they form the basis for the rest of the paper.

In visual algorithm simulation, the user manipulates graphical objects on the screen that are visual representations of actually implemented underlying data structures. Typically a simulation sequence consists of a number of context sensitive drag & drop operations, each simulating, *e.g.*, basic variable assignments, reference manipulations, or operation invocations such as insertions and deletions. The system interprets the operations and modifies the corresponding underlying data structures, such as arrays, lists or trees according to the operations, and automatically updates the visual representation on the screen. Thus, visual algorithm simulation applies automatic algorithm animation to update the screen. The conceptual difference between algorithm animation and algorithm simulation is that in the animation all changes in data structure representations are based on the execution of a predefined algorithm whereas in the simulation, the user is the active part initiating the changes. As a whole, a seamless combination of these methods, such as introduced in the Matrix application framework (Korhonen and Malmi, 2002), allows the user to explore the working of different algorithms and interact with the system in many ways.

If compared with plain algorithm animation tools, the simulation facility enables us to define new interesting types of algorithmic exercises, because the user-generated simulation sequence can be compared with a sequence generated by a true implemented algorithm. Thus, we can build exercises that train and test different aspects of the working of algorithms. We

[1]The research pages for TRAKLA2 learning environment at http://www.cs.hut.fi/Research/TRAKLA2/ includes fully working applets that demonstrate the algorithm simulation exercises.

can request the user to imitate a real algorithm with a given initial data, as depicted in Figure 1. Or, we can ask the user to solve a counter example: for a given algorithm, generate the initial data that produce the given output. There are also other possibilities as we can see in the next section.


TRAKLA2 - Binary Search Tree Insertion - Microsoft Internet Explorer

http://www.cs.hut.fi/Research/TRAKLA2/exercises/BST_Insert.html

Index TRAKLA2 Research TRAKLA2 Support

Binary Search Tree Insertion

Insert the given keys into the initially empty binary search tree. The keys can be inserted by drag & dropping them from the Stream of Keys into the correct positions in the tree below. If there are equal keys they are always inserted into the left branch of the existing node.

Applet applications.exerciseapplet.ExerciseApplet started

Figure 1: TRAKLA2 applet. The exercise window includes the data structures and push buttons. The model solution window is open in the front.

In the figure, we show a sample exercise in which the learner is to insert (drag and drop) the keys from the given array (initial data) one by one into the binary search tree. After completing the exercise, the learner can ask the system to grade his or her performance. The feedback received contains the number of correct steps out of the maximum number of steps. The learner can also view the model solution for the exercise as an algorithm animation sequence. The model solution is shown as a sequence of discrete states of the data structures, which can be browsed backwards and forwards in the same way as the learner's own solution.

3 Taxonomy

As the dynamics of a simulation is targeted to algorithms, we can derive the taxonomy of algorithm simulation exercises by examining the function $P : I \rightarrow O$ that an algorithm A is supposed to compute. An *algorithm simulation exercise* can employ any of the following three components, *i.e.*, the algorithm A (instructions to compute P), the input I or the output O, or any combination of them, while the other components are fixed (predefined in the assignment). Thus, an *algorithm simulation exercise* is a tuple $E_P = (A, I, O)$, where P

is the fixed problem to be solved by an algorithm A, I is the legitimate input set (a problem instance), and the solution is the obtained output set $O = A(I)$.

An *algorithm simulation exercise type* is a tuple $E_T = (X_1X_2X_3)$, where X_i is substituted by F if the corresponding E-component, *i.e.*, algorithm, input or output is fixed, respectively, and Q if it is in question. Most of the TRAKLA exercises ask the student to simulate a particular algorithm A with individually tailored input sequence $f_1, f_2, \ldots, f_k \in I$, and show how the corresponding structures change by determining the output sequence $q_1, q_2, \ldots, q_k \in O$. Thus, the exercise is of type $E_T = (FFQ)$ and we denote the input and output sequences as F_k and Q_k, respectively. The question is, in general, what is the output for the given algorithm with the given input. For example, F_k can be an ordered set of keys to be inserted into an initially empty binary search tree or it can be the parameter(s) an algorithm receives in each recursive call. Here, for each f_j the corresponding $q_j = A_{q_{j-1}}(f_j)$, *i.e.*, the binary search tree after each insertion or the computed result after each recursive call to A.

Two different subtypes for E-components can be identified: implicit and explicit. An *explicit question* (denoted by capital case Q) requires the learner to produce an answer for the question, for example, the output in the previous example. *Implicit questions* (denoted by lower case q) only require that the learner is familiar with the topic in question but no explicit answer is required and the learner may choose from among a set of alternatives. For example, the learner may be asked to apply any algorithm that produces topological sorting and not any particular one. Usually, this happens when there is more than one Q-component in an exercise. On the other hand, also fixed E-components can be implicit. For example, it is obvious what is the output structure while sorting an array of keys. Thus, we denote the implicit output as lower case f.

Eight different basic types of exercises can be named and characterized and 256 in total if the subtypes are taken into account. In the following, we summarize the 8 basic types briefly and give an example of each that includes also the subtype definitions.

1. $E_1 = (FF_if)$ - *Determining characteristics*: Which items in the array F_i are compared with the given search key $k \notin F_i$ in binary search?

2. $E_2 = (F_aF_iQ)$ - *Tracing exercise*: i) Insert the set of input keys F_i into an initially empty binary search tree. ii) Insert the set of keys F_i into an initially empty hash table using linear probing with the given hash function F_a.

3. $E_3 = (FQF_o)$ - *Reverse engineering exercise*: Determine a valid insertion order for the keys resulting the AVL tree F_o in question.

4. $E_4 = (FQq)$ - *Exploration*: Determine such an input string for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution.

5. $E_5 = (QF_iF_o)$ - *Determining algorithm*: The following binary tree F_i was traversed in different orders. The resulting traversing order of nodes are F_o. Name the algorithms.

6. $E_6 = (qFQ)$ - *Open tracing exercise*: i) Trace topological sort on a given graph. ii) Compare several recursive sorting algorithms with each other. Show the state of the input array F after each recursive call.

7. $E_7 = (qQf)$ - *Open reverse engineering exercise*: Compare several sorting algorithms to each other. Determine an example input for each of them that leads to the worst case behavior.

8. $E_8 = (QQQ)$ - *Completely open question.* Let us consider the following broken binary search tree. All duplicate keys are inserted into the left branch of the tree, but the deletion of a node having two children replaces the key in the node with the next largest

item (from the right branch). Give a sequence of insert and delete operations with keys of your choice that results in a tree that is no longer a valid binary search tree (Hint: the search routine can only find duplicates from the left branch).

Obviously the taxonomy presented here is not exhaustive in the sense that there exist exercises that are not algorithm simulation exercises at all. However, the taxonomy gives us a systematic way to cover one particularly interesting area of exercises throughout, and consider how such exercises could be supported by visualization tools.

4 TRAKLA and TRAKLA2 exercises

TRAKLA (Hyvönen and Malmi, 1993; Korhonen and Malmi, 2000) was initially implemented in 1991 to assess manual algorithm simulation exercises, *i.e.*, the learners solved the exercises on pen and paper and submitted the solution to the TRAKLA server by email. Later on, a visual front end for editing the answer was added, but there was no change in the system capabilities, because the front end was a dummy drawing tool with no understanding on the underlying data structures. TRAKLA2 (Korhonen et al., 2003), on the other hand, is a web-based learning enviroment built on the Matrix framework that provides full support for visual algorithm simulation and automatic algorithm animation.

We summarize the types of exercises that TRAKLA and TRAKLA2 systems support in Table 1. We use the systems as a proof of concept to address the many possibilities that automatically assessed algorithm simulation exercises have in the broad context of data structures and algorithms.

Table 1: Examples of automatically assessed exercises supported by TRAKLA and TRAKLA2.

Exercise type	TRAKLA2	TRAKLA
$E_1 = (FFF)$	FF_if	FF_if
$E_2 = (FFQ)$	F_aF_iQ	F_aF_iQ, FF_iQ, fF_iQ
$E_3 = (FQF)$		
$E_4 = (FQQ)$	(FQq, Q implies q)	
$E_5 = (QFF)$		
$E_6 = (QFQ)$		qF_iQ, q implies Q
$E_7 = (QQF)$	(qQ_if)	
$E_8 = (QQQ)$	$QQ'q, Q'$ implies q	

All the exercise types marked for TRAKLA and exercise types FF_if, FF_iQ,and $QQ'q$ in TRAKLA2 have been in production use with hundreds of students. For the FQq exercise, we have a reference implementation. For the tree traversing exercises ($E_5 = (QF_iF_o)$) mentioned on page 121 and Huffman code exercise ($E_6 = (qF_iQ)$) introduced with TRAKLA, however, we have a little bit different scheme. They will be implemented as a tracing exercise ($E_2 = (FF_iQ)$), and an open reverse engineering exercise ($E_7 = (qQ_if)$), respectively. It should be noted, however, that also exercise types not marked for TRAKLA2 can and will be incorporated into the system in the future, but for brevity these are not discussed here any further. In the following, we map each example exercise type described above to an actual exercise implemented or designed.

1. FF_if; binary and interpolation search; the algorithm is explicitly determined as well as the parametrized input. The result is obvious as the key is not found from the structure. The user must determine which items in the array F_i are compared with the key to be searched during the search.

2. F_aF_iQ, FF_iQ, fF_iQ; linear probing, deque, radix sort; the algorithm can be parametrized as it is in case of linear probing, it can also be expressed implicitly as it is done on radix sort (the learner must determine which radix sort is simulated by examining the intermediate states of the input structure).

3. FQq (TRAKLA2 only); BMH-algorithm; determine such an input Q for Boyer-Moore-Horspool algorithm that satisfies the statement coverage (every statement is executed at least once with the test set). Describe the output of such an execution. The selected input Q implies the output q.

4. qF_iQ, q implies Q; topological sort, Huffman code; in both of these exercises several correct solutions are possible for a given input. The algorithm applied by the learner implies the resulting output.

5. qQ_if; Huffman code; the learner is asked to simulate Huffman's algorithm q to form the Huffman code f in order to decode the original text string Q_i. The actual visible input for the algorithm is the frequencies of the characters in Q_i.

6. $QQ'q, Q'$ implies q; broken binary search tree; many procedures can lead into a correct solution. Several correct answers are possible due to the nature of the exercise. The input keys chosen by the learner imply the resulting output.

As mentioned in the introduction, there are few other systems that support algorithm simulation exercises. PILOT (Bridgeman et al., 2000) is targeted to tracing exercises that employ graph algorithms. There is also an option to allow parametrized input graphs. Thus, the type of the exercises is $E_2 = (FF_iQ)$. On the other hand, "stop-and-think" questions requiring an immediate response from the learner introduced in JHAVÉ (Naps et al., 2000) can be interpreted to be $E_1 = (FF_if)$ questions where the learner determines characteristics of the given algorithm. Moreover, the same system illustrates exercises in which the learner is asked to manually trace an algorithm on a small data set. However, in this case, there is no automatic assessment involved.

5 Discussion

The taxonomy of algorithm simulation exercises was first presented by Korhonen (2003). However, an interesting analogy exists independently within a completely different area of teaching. In their paper, Sutinen and Tarhio (2001) present a similar example of problem classes that are applied for characterizing problem management in thinking tools. They also have three components (Start (input), Technique (algorithm), and Goal (output)) that they call dimensions. Again, these expand to eight classes, if restricted to binary values "open" and "closed" (in question and fixed above). This construction spans the creative problem management space.

Such a problem management space is useful for teachers in various ways. Regardless of whether we are designing a single course or a larger program, we can use this space as a reference to better identify what are the learning goals of the exercises. Typically the exercises change from closed exercises to more open ones, when students progress in their studies. In closed exercises students generally train specific skills and techniques whereas in open exercises they have to apply their knowledge to solve new and varying problems. However, the presented problem space provides for a broader view of this issue than such a one-dimensional closed – open axis.

Let us now consider more closely, how this applies to a data structures and algorithms course. When learning this topic, students first have to learn how various well-known algorithms, such as quicksort and binary search algorithms work. These can be trained with exercises of types FFF and FFQ (and possibly qFQ). Next, they should understand the behaviour of the algorithms. Typically this is strongly related to mathematical analysis of presented algorithms, especially considering their worst-case behaviour. Attached to this theme, common assignments deal with generating worst case or best case input data for a given algorithm (type FQf), or more generally compare worst cases / best cases of a set of algorithms (type QQf).

In traditional education of the topic there are, however, seldom exercises in which students could explore the behaviour of the algorithm using different kinds of input sets. The obvious reason is that this may be clumsy on pen and paper, or laborious, if it requires coding. On the other hand, using visual algorithm simulation such exercises are easy to organize. Moreover, providing automatic feedback on the answers is often simple, as well. As an example, a student could be given an FQq type of exercise: "Insert the following keys into a red-black tree in such order that the height of the resulting tree is at least 6", or an FQF type exercise "Insert the following keys into a binary search tree in such order that the resulting tree is a complete binary tree". Another exercise of type FQq is "Consider the following weighted graph. You may modify the weight of at most 3 edges to create a graph in which both Prim's and Djikstra's algorithms starting at node A create the same spanning tree. Is this possible?". Note that in such exercises it is not enough just to apply the algorithm. The student has to recognize how the result is affected by changes in the input data, which is something else than understanding the worst-case or best-case behavior of the algorithm. Still, automatic assessment of the solution is straightforward.

Constructing new algorithms to solve new problems is a standard type of exercises in an algorithms course. Such exercises are of type QFF or QFq. If we allow the student freely demonstrate his/her skills on the course theme, we end up in QQQ type of totally open exercises, even though we can limit the exercise topic as we did in the broken binary search tree exercise.

In the previous discussion, we have provided classifications for existing exercises. Conversely, the teacher can try to devise exercises of each type in the taxonomy, to find out new ways of handling the topic. Especially exercises which are related to exploring the behaviour of the algorithm (FQQ, FQF) can enrich the commonly used exercise sets. The taxonomy thus promotes creating new course material.

Next, we continue to discuss some other aspects of the taxonomy. Two main characteristics should be taken into account while designing new algorithm simulation exercises. First, we should make clear how these exercises are delivered to the students. We identify two main classes here, namely *closed labs* and *automatically assessed exercises.* The main difference between these two is the available support from instructors. In closed lab sessions, we assume that an instructor is present and takes actions to guide the learner by asking specifying questions, giving additional feedback, and so on. The exercises can be solved by "exploring" the state space and the correctness of such an exploration can be assessed by the instructor. Thus, the exercises are more open in their nature. On the other hand, automatically assessed exercises should be self-explanatory so that the learner can cope with the assignments by himself. In addition, the feedback should be explicitly targeted to the assignment in question and aid the learner to find the correct solution. Roughly speaking, the most open questions (types QFQ, QQF, QQQ) are more suitable for closed labs while tracing exercises (FFF, FFQ, FQF) suit well to automatic assessment. There are, however, counter examples that contradict this discrete view.

Finally, we mention an important characteristics (especially with automatically assessed exercises) in which the fixed E-components can be parametrized (denoted by subindex) so that each learner has his own individualized exercises. For example, in the tracing exercise $E_2 = (FF_iQ)$, where keys are inserted into an initially empty binary search tree, the keys F_i to be inserted can be randomly drawn from the set of all possible keys. Thus, each learner has an individually tailored set of keys to be inserted. This is the approach used both in TRAKLA and TRAKLA2 although they have slightly different policies concerning resubmission of solutions. In TRAKLA exercises, the learner is allowed to resubmit his/her solution to a particular exercise to the server only a few times, and each time the initial data remains the same. In TRAKLA2, on the other hand, the initial data is randomized each time the learner wants to continue after requesting grading or the model solution, but then there is no limit for the number of grading requests.

In the case of input and output components, the randomization is trivially achieved by randomly picking a suitable number of keys into the corresponding structure. Of course, some constraints must be taken into account in order to prevent the exercise turning out to be trivial (for example, the AVL tree insertions should include both single and double rotations). However, parametrized algorithm exercises $E_{1...4} = (F_a XX)$ are slightly trickier. Usually, the implementation of such a parametrization depends on the algorithm. For example, in linear probing, the hash function $h(k) = (k+p) \bmod q$ can be parametrized by individually tailoring p and q.

6 Summary

We have presented a novel method for classifying algorithm simulation exercises. The presented taxonomy can be used not only as a classification tool, but also as a design tool for teachers when they create new exercises for their data structures and algorithms courses. The taxonomy can give more insight into this design process, and as such aid teachers to refine the learning goals they wish to set up for their students. It seems that in most cases the exercises can be supported with automatic feedback on students' answers. Our TRAKLA2 has demonstrated this widely with tracing exercises, but we have a lot interesting work to do to implement new types of exercises in different fields of algorithms.

Finally, we hope that our taxonomy will promote other systems to implement algorithm simulation, and apply these ideas in other environments, as well.

References

S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: An interactive tool for learning and grading. In *The proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 139–143. ACM, 2000. URL `citeseer.nj.nec.com/bridgeman00pilot.html`.

Juha Hyvönen and Lauri Malmi. TRAKLA – a system for teaching algorithms using email and a graphical editor. In *Proceedings of HYPERMEDIA in Vaasa*, pages 141–147, 1993.

Ari Korhonen. *Visual Algorithm Simulation.* Doctoral thesis, Helsinki University of Technology, 2003.

Ari Korhonen and Lauri Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, 2000. ACM.

Ari Korhonen and Lauri Malmi. Matrix — Concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–114, Trento, Italy, May 2002. ACM.

Ari Korhonen, Lauri Malmi, and Panu Silvasti. TRAKLA2: a framework for automatically assessed visual algorithm simulation exercises. In *Proceedings of Kolin Kolistelut / Koli Calling – Third Annual Baltic Conference on Computer Science Education*, pages 48–56, Joensuu, Finland, 2003.

Aravind K. Krishna and Amruth N. Kumar. A problem generator to learn expression: evaluation in CSI, and its effectiveness. In *Proceedings of the sixth annual CCSC northeastern conference on The journal of computing in small colleges*, pages 34–43. The Consortium for Computing in Small Colleges, 2001.

Thomas L. Naps, James R. Eagan, and Laura L. Norton. JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. In *Proceedings of the SIGCSE Session*, pages 109–113, Austin, Texas, March 2000. ACM.

Erkki Sutinen and Jorma Tarhio. Teaching to identify problems in a creative way. In *Proceedings of the 31st ASEE/IEEE Frontiers in Education Conference*, page p. T1D813. IEEE, 2001.