Selecting a Visualization System

Sarah Pollack, Mordechai Ben-Ari
Department of Science Teaching, Weizmann Institute of Science, Israel

moti.ben-ari@weizmann.ac.il

1 Introduction

This paper describes the selection of an algorithm visualization system for use in a high-school course on data structures. While the selection criteria and the evaluations of the systems are highly specific to the particular educational context, we believe that it will be helpful to share our experience with others. Even more important, we wish to point out areas in which *none* of the systems fulfilled our requirements, so that developers of existing or future visualization systems can take them into account.

1.1 The educational context

During the 1990s, a new curriculum was developed for Israeli high schools (Gal-Ezer et al., 1995). The intent was to enable students to study computer science as a full-fledged scientific subject for their matriculation examinations. These subjects can be studied in three or five 90-hour *units* spread over three years of high school. The five-unit curriculum includes three required units: a two-unit course on the foundations of computer science and a unit called *Software Design*, as well as two elective units, one theoretical and one on a second programming paradigm. The Software Design unit is normally studied in the last year of high school; its syllabus includes the study of (a) designing non-trivial programs and implementing them in modules, and (b) data structures (lists, stacks, queues, trees) and elementary algorithms on these structures.

We have had experience in the design and development of the program animation system Jeliot (Ben-Ari et al., 2002). Our research has shown that visualization can improve the learning of programming by novices by providing them with a concrete vocabulary for describing the execution of programming constructs (Ben-Bassat Levy et al., 2003). However, Jeliot visualizes the lowest level of program execution (individual variables, expression evaluation and instruction execution), and thus is not suitable for use in the Software Design course, where we need to visualize more abstract entities like lists and trees. This paper describes our evaluation of visualization systems to support learning and teaching in this course.

1.2 The visualization systems evaluated

Based upon our previous acquaintance with visualization systems, we chose Animal (Rößling and Freisleben, 2002) and Matrix (Korhonen and Malmi, 2002) as the candidate systems. Later, we also decided to evaluate Interactive Data Structure Visualizations (IDSV) (Jarc, 1999).

Animal is an interactive system for building visualizations and animations. It is an open tool, meaning that there are no data structures and algorithms predefined by the system; instead, the tool supplies graphics primitives that are useful in creating such visualization. (A library of visualizations of data structures and algorithms is available.) In addition to its interactive mode, a scripting language (Rößling and Freisleben, 2001) is available; this provides an alternate method of generating visualizations (one that may be faster in the hands of an experienced user) and it enables the construction of meta-tools to generate visualizations. Such a tool is available for generating visualizations of sorting algorithms. Animal can be downloaded from `http://www.animal.ahrgr.de/`.

Matrix is a system for visualizing data structures and algorithms, primarily, advanced tree and graph algorithms. It is closed, meaning that there is a fixed set of structures and

algorithms supported by the system. We evaluated the most recent version of the system called MatrixPro that can be downloaded from `http://www.cs.hut.fi/Research/MatrixPro/`.

IDSV is a web-based client-server tool that integrates HTML documents with animations of data structures and algorithms. IDSV can be run interactively from `http://nova.umuc.edu/~jarc/idsv/`. IDSV is also a closed tool, offering a limit number of visualizations, but the documents can be easily changed from outside the tool. In our environment, we can assume that schools (and students' homes) are reasonably well-equipped with computers, but we cannot take for granted the existence of good Internet access. By courtesy of the developer, we received the source code of IDSV and created a self-contained Java application suitable for our environment.

2 Selection criteria

If there were only one set of criteria for choosing a visualization, there would be only one visualization system. Clearly, the multiplicity of systems means that different researchers and educators have different sets of criteria. In this section, we set out the criteria that we used in our selection.

The first and most important criterion is the appropriateness of the visualization for the intended users—high-school teachers and students—and the probable contribution to learning from use of the visualization system. The users can be characterized as having very good computer literacy skills, and beginning programming skills in the language used in the course (Pascal or C). Therefore, ease of installation and use is a primary consideration, and the effective use of a visualization tool *cannot* require proficiency in Java programming. (All three systems are written in Java and we had no problems installing or using them.) There are plans to use Java and C# in Software Design course in the future, but we can never expect teachers and students to acquire the skills to effectively modify large Java programs.

Here is a list of aspects of the tools that were assessed under this criterion:

- Ease of installation and operation.
- Support for the teacher to easily create demonstrations.
- The ability to integrate the tool within the type of teaching activities that the teachers routinely use. (Teachers tend to be conservative in adopting new educational technology, especially if it affects their existing corpus of lesson plans and teaching materials.)
- Beyond the basic visualization of the data structures and algorithms, the tool should enable the students and teachers to analyze data types, for example, to analyze the efficiency of a tree search in relation to the height of the tree.
- The visualization tool should support communications between the teacher and student: Can the student use the tool to ask questions and can the teacher use the tool to answer them?

The second criterion is that a visualization tool must support visualizations of the data structures and algorithms actually used in the course, preferably through built-in visualizations that will not require extra effort on our part. The algorithms to be studied include creation of the data structure, adding and removing elements, and searches and traversals. In order to support the creation of exercises for students, it is essential that the tool have the ability to easily input data sets.

The third criterion is the quality of the visualization. We looked for the ability to control the speed of animation, to choose step-by-step or continuous execution, and to step backwards and forwards. Control of the step granularity is very helpful and it is important that the visualization be coordinated with a display of the source code (whether in a programming language

or pseudocode). A save/load facility is essential to help teachers prepare visualizations for demonstrations.

In terms of the criteria given by Anderson and Naps (2000), we want Level I (understanding of the algorithm as a recipe) and Level 2 (understanding the relationship of the algorithm to its implementation) of their Algorithm Understanding Scale. For their Instructional Design Scale, the important levels are Level 1 (the visualization ... demands no interaction from the viewer), and Level III (allow the student to design input data for the algorithm). Flexibility of the visualization is less important, because the students are young novices, and the presentation of textual material is not too important, because we intend to supply our own material tailored to contents of the course.

3 Evaluation

The evaluation was carried out by the first author, who is a very experienced teacher of computer science in high schools; in particular, she has taught the Software Design course for many years. Four typical exercises using binary search trees (BST) were written in the style that such exercises would be given to students or used by a teacher for demonstration: constructing a BST, adding an element to a BST, searching for a value in both balanced and unbalanced BSTs, and using a BST for sorting. The tools were then analyzed to determine their potential for improving the learning experience when used with these exercises. A table was created for each of the criteria and sub-criteria, and scores and weights were assigned.

3.1 Animal

As an interactive tool ANIMAL is very easy to use, because its intuitive drag-and-drop interface makes it possible to produce new visualizations with little training. However, the amount of time required to create a visualization is quite large. While investing this amount of time can be justified when preparing a large formal lecture, it would not be practical in a high-school classroom. The classroom dynamics is such that a teacher should be able to use the visualization to quickly respond to questions such as: "What would happen if the last value to be inserted in the tree were 15 and not 30?" Therefore, the tool received a low score for the criterion of aiding communication between the student and the teacher.

The problem is that the ANIMAL tool does not contain algorithm-specific knowledge. While it does supply primitives that facilitate the creation of algorithm animations, we could not see that teachers and students would find these sufficiently more powerful than those available in a generic tool like PowerPoint, with which they are very familiar.

The attractive mode of the use of ANIMAL is to use meta-tools to generate ANIMALSCRIPT for algorithm visualizations. Meta-tools enable the data set to be easily changed for exercises. The ANIMAL web site includes such a tool for sorting algorithms, but these algorithms are covered in the second year course, not in the Software Design course. Building meta-tools is not difficult, but it is not feasible given the low programming skills of the teachers and students. The option of manually writing ANIMALSCRIPT commands is not an attractive option for a target audience who are used to working with interactive tools.

The ANIMAL library includes many contributed visualizations, which would be invaluable in preparing lectures, but the visualizations are static in the sense that to change them to use different data requires that they be entirely recreated. Therefore, it would be difficult for the teacher to integrate the visualizations within the existing activities in the course.

The quality of the visualizations and the control that the user has is good. The visualization can be run step-by-step or it can be displayed as a smooth animation. A unique advantage of ANIMAL is that its visualization primitives support the display of source code, and you can (manually) coordinate between the code and the visualization. Finally, ANIMAL contains no support for assessment.

3.2 Matrix

Matrix comes with a large built-in selection of algorithm visualizations, mostly for advanced algorithms. The developers kindly agreed to add visualizations of simple data structures like stacks and queues, so that we have a set of visualizations covering the data structures taught in the course.

Matrix uses step-by-step animation, though it can export the animation in the Scalable Vector Graphics (SVG) format and these animations are smooth. The control of the visualization is good and animations can be saved and reloaded. Of particular importance is the ability of Matrix to read external files of data so that the teacher can easily prepare different demonstrations and exercises for a particular algorithm, for example to show the dependency of the efficiency of a BST search on the balance of the tree. Since the files are text files, they could even be modified during a class using an editor. Matrix also includes a self-assessment facility.

The control of the animation is excellent: not only can you run the animation forwards or backwards, and step-by-step or continuously, but you can also define breakpoints and specify the granularity of the animation step.

Matrix has the novel ability to *simulate* algorithms (Korhonen, 2003). In algorithm simulation, the user can manually modify the visual representation of the data structure, and these modifications are applied to the internal representation of the structure. We have not yet decided if this feature is important in our context.

The most serious problem with Matrix is that there is no display of source code nor is there any coordination with explanatory material. It is also difficult to extend it to visualize new algorithms. Such an extension was demonstrated to us by the developers, but it requires significant Java programming experience and familiarity with the structure of the source code, both of which are way beyond the capability of high school teachers.

3.3 IDSV

We modified the original IDSV tool to be a Java application using the Swing user interface; source code and explanations are displayed from text files distributed in the tool archive. However, the source code is just displayed, not coordinated with the animation of the algorithm.

IDSV uses smooth animation, where you can see an element slowly travel throughout the data structure, rather than just appear in its correct position. Based upon our experience with Jeliot, we believe that smooth animation is better for beginning students learning elementary algorithms. Furthermore, like Jeliot, IDSV displays helpful comments at each step. Since these features proved to be extremely useful in teaching introductory students, this made IDSV extremely attractive for our high schools students.

The control of the animation is less complete than in the other tools, and you cannot run animations backwards.

There are two main problems with IDSV. First, there is a limited selection of built-in data structures and algorithms, and like with Matrix, adding new visualizations requires significant effort and Java programming experience. Second (and more important), while IDSV enables the user to see animations run with random data or data entered interactively, there is no way to supply a file with a data set.

4 Our selection

The three tools were evaluated according to a set of criteria and weighted numerical scores were given and justified.[1] Data adapted from that report are shown in Table 1. More important

[1]The full report—in Hebrew—is available from the authors.

Criterion	Subcriterion	Weight	Subweight	Matrix	IDSV	Animal
Usability		20%		5	3	3.8
	Installation		20%	5	5	5
	Existing animations		40%	5	3	5
	Creating animations		40%	5	2	2
Visualization		30%		3.7	2.8	3.1
	Visualization of ADT		10%	5	2	3
	Control of animations		35%	5	3	5
	Coordinated with algorithm		30%	1	4	5
	Reinitialize ADT		30%	5	3	1
Pedagogy		50%		5	2.75	1.75
	ADTs and their operations		25%	5	3	3
	Open to new ADTs		25%	5	3	2
	Building exercises		25%	5	2	1
	Learning activities		25%	5	3	3
Total		100%		4.6	2.8	2.6

Table 1: Weighted scores of the visualization tools

than the actual scores, however, the match between the capabilities of the tools and the needs of our students and teachers.

We found that ANIMAL supports the visualization of the data structures that are taught in the course, but it is difficult to integrate into the type of activities that are currently used. Furthermore, neither of the modes of the use of ANIMAL—interactive and scripts—is appropriate for our environment.

In deciding between IDSV and Matrix, IDSV has the advantages of smooth animation and display of code and explanations, while Matrix has a larger set of built-in visualization and better control of the visualization. We decided to adopt Matrix, primarily because of its excellent flexibility: the ability to save and load animations, and to create data sets for exercises and examinations. We hope that future versions of Matrix will improve in the two areas in which it is deficient: display of source code and ease of extension.

5 The problem with all of them

The emphasis in the Software Design course for high-school students is on understanding the concept of abstract data type (ADT), building modules to implement ADTs and using them to solve problems. Here is a list of typical problems that would be given as exercises on homework or examinations:

- Let `L1` and `L2` be ordered lists of integers. Write an algorithm that returns the list resulting from the removal of all elements of `L1` from `L2`.
- Write an algorithm for the boolean-valued operation `immediately-after(L,x,y)`, where `L` is a list of integers and `x` and `y` are integers; the operation returns `true` iff `x` appears in `L` immediately after `y` or `y` appears in `L` immediately after `x`.
- Write an algorithm to compute the number of nodes in a binary tree.
- Write an algorithm to check if two binary trees are "mirror images" of each other.
- Define a *sum-tree* as a binary tree such that the value at every node is larger than the sum of all the values of the nodes in the left subtree and smaller than the sum of the

values of the nodes in the right subtree. Write algorithms to create a sum-tree and to check if a tree is a sum-tree.

With closed systems like Matrix and IDSV, we cannot create animations for these problems. With an open system like ANIMAL, any animation can be created, but this is a task that is separate from the task of solving the problem. To build an animation, a student would first have to solve the problem, but once she solves the problem, there doesn't seem to be any point in putting in the extra effort to create the animation.

We believe that if *student-written* algorithms could be easily animated, it would improve the students' ability to successfully solve problems. Ideally, we would want a visualization system to be able to work directly from student-written source code, as is done in Jeliot (Ben-Ari et al., 2002).

The objection that is made to self-animation is that it cannot be as good as animations that are hand-crafted for particular algorithms. While this is true to some extent, courses in data structures and algorithms study a very limited number of generic structures: arrays, lists, trees, graphs. (These are called *fundamental data structures* in Matrix (Korhonen, 2003).) It ought to be possible to have a visualization system interpret source code manipulating these structures. The feasibility of this approach was demonstrated in the first version of Jeliot, which animated array, stack and queue algorithms by modifying the source code with no further user intervention.

A related issue that we would like to see addressed is that of visualization of self-assessment facilities. As implemented in Matrix and IDSV, the student simply receives a score and the correct answer. What we would like to see is a visualization that would point to a node and display "the key 5 is less than the value 10 in *this* node, but you entered the right subtree instead of the left subtree."

6 Other systems

There are other visualization systems that are worth evaluating, but that were not appropriate for our study because they are intended for teaching that uses the Java programming language. Two of them are worth mentioning, because they address issues discussed in this paper.

Dot.java (Hamer, 2004) is a Java class that can be included in a student's program. It provides a method `drawGraph` that creates a drawing of any Java object. The drawing is visualized with the GraphViz utility. Although it requires intervention in the source program, the intervention is very simple. The primary advantage of Dot.java is that it enables visualization of *any* student-written program, which we believe to be of great importance.

jGrasp (Hendrix et al., 2004) is a well-known pedagogical development environment. The latest version contains the beginnings of a dynamic visualization feature. The advantage of here is that the visualization is integrated into an IDE so a separate tool is not required.

7 Conclusions

The choice of Matrix was primarily dictated by the flexibility it offered for building exercises that would enable the analysis of algorithms. We were not aware of Karavirta et al. (2002) when performing our comparison, so it is gratifying to see that our analysis is consistent with the concepts and analysis presented in that paper. Given the level of the students and teachers it is essential that the visualizations be as *effortless* as possible. Figure 1 of Karavirta et al. (2002) shows that ANIMAL is generic and high effort, whereas Matrix is specific and low effort; similarly, Figure 2 ranks ANIMAL as having a primitive graphical vocabulary and high effort, whereas Matrix has a complex graphical vocabulary and is low effort. (Note that the scripting language of ANIMAL was not evaluated in this study.)

We would like to see tool developers explicitly discuss the intended pedagogical use of their tools; an example of such an explicit discussion is given in Chapter 9 of Korhonen (2003). We

believe that a priority for future development of algorithm visualization systems is to enable the visualization of student-written algorithms.

Acknowledgements

We would like to thank the developers of the visualization systems for their willingness to answer our questions. This research was partially funded by the Israeli Ministry of Education.

References

Jay Martin Anderson and Thomas L. Naps. A context for the assessment of algorithm visualization systems as pedagogical tools. In *Proceedings of the First Program Visualization Workshop*, pages 121–130, Porvoo, Finland, 2000.

Mordechai Ben-Ari, Niko Myller, Erkki Sutinen, and Jorma Tarhio. Perspectives on program animation with Jeliot. In *Software Visualization: International Seminar*, Lecture Notes in Computer Science 2269, pages 31–45, Dagstuhl Castle, Germany, 2002.

Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka A. Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1–15, 2003.

Judith Gal-Ezer, Catriel Beeri, David Harel, and Amiram Yehudai. A high school program in computer science. *IEEE Computer*, 28(10):73–80, 1995.

John Hamer. A lightweight visualiser for Java. In *Proceedings of the Third Program Visualization Workshop*, pages 54–61, Warwick, UK, 2004.

T. Dean Hendrix, James H. Cross II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in an lightweight IDE. *SIGCSE Bulletin*, 36(1):387–391, 2004.

Duane J. Jarc. *Assessing the Benefits of Interactivity and the Influence of Learning Styles on the Effectiveness of Algorithm Animation Using Web-based Data Structures Courseware.* PhD thesis, George Washington University, 1999. `http://www.student.seas.gwu.edu/~idsv/djj-dissertation.pdf`.

Ville Karavirta, Ari Korhonen, Jussi Nikander, and Petri Tenhunen. Effortless creation of algorithm visualization. In *Proceedings of the Second Finnish / Baltic Sea Conference of Computer Science Education*, pages 52–56, 2002.

Ari Korhonen. *Visual Algorithm Simulation.* PhD thesis, Helsinki University of Technology, 2003.
`http://lib.hut.fi/Diss/2003/isbn9512267950/isbn9512267950.pdf`.

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, 2002.

Guido Rößling and Bernd Freisleben. AnimalScript: An extensible scripting language for algorithm animation. *SIGCSE Bulletin*, 33(1):70–74, 2001.

Guido Rößling and Bernd Freisleben. Animal: A system for supporting multiple roles in algorithm animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.