Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 27 (2014) 229 – 239

Procedia Computer Science

5th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion, DSAI 2013

Design and evaluation of a web-based dynamic algorithm visualization environment for novices

Euripides Vrachnos, Athanassios Jimoyiannis*

Department of Social and Educational Policy, University of Peloponnese, Korinthos, Greece

Abstract

Teaching basic algorithmic concepts to novices is not an easy task. Existing research has given considerable information about students' alternative conceptions and faulty mental models about abstract programming concepts and constructs, as well as their difficulties in solving programming problems. Various algorithm visualization systems are proposed as alternative and efficient instructional environments for introductory programming courses. They include dynamic features, based on animation techniques, aiming at illustrating the behavior of basic algorithms and fostering students' experimentation and algorithmic knowledge construction. This paper presents DAVE, a web-based dynamic algorithm visualization environment designed to support secondary education students' learning about basic algorithms. DAVE facilitates students' experimentation with array algorithms by allowing the modification of both code and data. The presentation of preliminary results, obtained from an evaluation study, provided evidence of the usability of the system and its potential to support students' development of efficient mental models regarding basic array algorithms.

Selection and peer-review under responsibility of the Scientific Programme Committee of the 5th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (DSAI 2013).

Keywords: Algorithms, visualization environment, novice programmers, web-based learning

1. Introduction

In the dynamically evolving 21st century era, ICT are increasingly considered as the critical factor in

* Corresponding author. Tel.: +0-030-2741074350; fax: +0-030-2741074990.
E-mail address: ajimoyia@uop.gr

Selection and peer-review under responsibility of the Scientific Programme Committee of the 5th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion (DSAI 2013).
doi:10.1016/j.procs.2014.02.026

establishing opportunities and conditions for an inclusive 'learning society' for all [1]. People who can understand and effectively use digital means are significantly empowered in terms of educational opportunities, professional development, and active participation in social and economic life. Academics, educators, researchers, and policy officers have acknowledged that students need to develop a wide range of digital knowledge and skills, beyond the simple notion related to the use of computers and the Internet, which also include critical thinking, information management skills, algorithmic thinking and problem solving skills.

In this context, algorithmic thinking and programming skills play a central role in computing education. Students typically need to become familiar with a great number of different algorithms and data structures. The ability to design an algorithm for a given problem is one of the most important and challenging tasks in computer science education. However, literature shows that novices face serious difficulties in using abstract programming concepts like data structures (array, graphs, lists) and lack the skills necessary to function abstractively, to consolidate an algorithm as a single entity, to comprehend its main parts and the relations among them, and to compose new algorithms by using their previous programming knowledge [2, 3, 4,5, 6].

Teaching and learning how to program is a challenging and complex task that has over time proved to be a universal problem for both students and teachers. It is a common conclusion that students have faulty or 'non-viable' mental models concerning programming constructs, objects, and methods while they exhibit poor performance in using elementary problem-solving strategies, even in simple concepts like assignment [5]. In addition, educators cited failure in introductory programming courses and/or disenchantment with programming as major factors underlying the poor student retention in computing degree programmes [7].

The typical programming environments and languages used in instruction have been constructed for developing software applications rather than for educational purposes. This is considered as a main source of barriers for novices in programming. In the last decade, there is a growing interest about using alternative instructional approaches and learning environments for introductory programming. A better alternative might be to use a programming language or environment specifically designed for educational purposes. The first type consists of microworlds, e.g. micro-languages or applications of the form of graphical environments in which the student can move an object (such as a robot) on a canvas using a predetermined set of commands; Logo, Karel, BlueJ, MicroWolds Pro, RoboLab, Scratch, BYOB etc. are popular environments of this type.

The second type of environments includes Algorithm Visualization (AV) tools and learning systems. These tools aim to visualize/simulate what happens during the execution of an algorithm, e.g. searching or sorting algorithms. Indicative examples of AV are Animal [8], Jawaa [9], Jeliot [10], JHavé [11], Trakla2 [12] and Alvis [13]. Those systems aim to demonstrate the fundamental operations of basic algorithms concentrating more on abstractions of their behavior than on low-level program structures. Over the last decade, visualization systems have been used as effective learning tools and received increasing interest from both educators and researchers. Most AV systems provided promising results regarding their potential to promote students' engagement and experimentation, and to support construction of mental models about abstract programming constructions and the logic of algorithms, and influence students' algorithmic thinking and development of problem solving skills [14, 15].

In the last years, the diffusion of HTML5 technology and the consequent capabilities of high-quality browser-based graphics have led to the development of fully web-based applications [16] executable on any platform or device, e.g. PCs and mobile devices such as tablets and smartphones. As browser is becoming the universal interface to a range of applications, web-based learning environments have a growing impact on education and learning. The web-based Dynamic Algorithm Visualization Environment (DAVE), presented in this paper, belongs to this new category of algorithm visualization systems. It was designed to support and enhance secondary education students' learning about algorithms. The system is an extension of a previous version, implemented in Java [17], with similar functionality and pedagogical features. The new version of DAVE is fully web-based and uses customized visualizations for a series of algorithms about arrays.

The paper begins with an overview of literature about algorithm visualization. The technological and pedagogical features, the architecture and the functionality of the system are outlined in detail. Preliminary evaluation results of DAVE, as the outcome of an instructional intervention in two public secondary schools in Greece, are also presented. The paper concludes that algorithm visualization promoted students' engagement and

helped them to acquire algorithmic knowledge, to manipulate algorithms and solve problems with arrays.

2. Background

Despite that arrays constitute a fundamental data structure in introductory programming curriculum, only a small amount of research has been directed to the investigation of students' mental models and programming difficulties with arrays [2, 6, 18, 19]. According to a survey of computing educators, loops and arrays are two of the three programming topics of major difficulty for novice students [2]. Du Boulay reported on students' confusion between an array index and its cell; they also have difficulties to deal with arrays that contain indices as array elements [19]. An unpublished authors' investigation in Greek secondary schools, using a sample of 102 students (K-12), confirmed the same misconceptions and learning difficulties; the majority of the students had faulty or incomplete models of the array concept which resulted in misconceptions and serious difficulties in solving simple algorithmic problems which demand the use of array data structure.

In general, there are two main categories of visualization systems in CS education: program visualization and algorithm visualization systems.

Program Visualization (PV) systems produce direct representations of programming structures and/or program execution phases (e.g., values of variables, internal program structures, method frames, data structures, objects etc.). Jeliot 3 [10] is a well-known PV system which visualizes Java programs; other contemporary systems, like Jype [20], UUhistle [21] and Online Python Tutor [16], visualize programs in Python.

However, the logic behind an algorithm cannot be revealed by just showing how the values of the program variables change. Students need proper graphical representations which fit better to their mental models about the execution of the particular algorithm. Algorithm Visualization (AV) systems aim to cover this need by visualizing abstract concepts and unfolding the underlying logic of the algorithm under study, thus helping students to construct multiple mental models, to interlink construct hierarchies and generalize problem-solving patterns. The terms static and dynamic algorithm visualizations are used in the literature in order to distinguish the degree of interactivity and students' experimentation with the visualization (e.g. abilities to modify both, input data and algorithm code, as well as various representations of the visual objects). A recent survey about algorithm visualization systems can be found in [22].

The first reference to algorithm animation was the famous video entitled 'Sorting Out Sorting' which has been presented by R. Baecker on 1981 at the SIGGRAPH Conference [23]. This 30 min video demonstrated the characteristics and the operations of nine sorting algorithms, using animation and audio comments. Since then, there were several tools developed as a result of research projects on algorithm visualization [14].

The most popular technique for creating an algorithm animation is by annotating the algorithm code with scripting commands producing the visualization. The first system based on a scripting language has been developed by John Stasko and his colleagues [24] and belongs to a wide family of algorithm visualization systems (Tango, Polka, Samba and JSamba). Animations consist of a file containing graphics instructions which correspond to important events of the algorithm under visualization.

Another family of systems, like MatrixPro [25], Trakla2 [12] and Ville [26], provide "Algorithm Simulation Exercises", where the student has to manually perform a given algorithm, typically by dragging elements to new or target positions or by clicking on buttons to cause a certain function. Ville is a new tool of this family; it supports multiple programming languages includibg C++ and Java. Its built-in editor supports creation of interactive quizzes and tests displayed as pop-up windows.

Another novel AV system is JHave [11] which helps AV developers to easily create animated slideshows. Its specific feature is the 'stop-and-think' questions and explanations that can appear at any time during the execution of the animation, thus promoting students' active interaction with the visualization of the algorithm. JHave includes a large collection of algorithm visualizations and has received great educational interest. A recent algorithm animation system is Alvis Live! [13]. It is a program development environment that supports construction and interactive presentation of algorithm visualizations using SALSA scripting language. It includes features that support story boarding. Moreover, Alvis Live! provides an error checking system, which reports error messages while the student develops his own code.

Several experiments have been conducted to investigate the educational value of algorithm visualization systems [27, 28, 29]. In overall, the results showed that simple animations or passive algorithm visualizations had minimal impact on students' learning, due to the low level of learners' engagement. Hundhausen, Douglas and Stasko performed a systematic evaluation regarding the pedagogical effectiveness of AV systems [27]. Their meta-study of 24 published investigations, related to AV, concluded that a) the way students use visualizations is more important than the visualizations themselves, and b) AV environments are effective only when the students are actively engaged into the learning process.

3. Design framework of DAVE

Literature suggested that effective algorithm visualization systems should be open and highly interactive to promote students' cognitive engagement [27,28,29]. To achieve a high level of learners' involvement, an AV system should allow students a) to modify preexisting algorithm visualizations, and b) to construct their own visualizations and reflect on them. No one of the presented AV systems allows students to modify the source code of a given algorithm and, consequently, to experiment with a visualization which is customized according to the inherent features of the algorithm. In addition, most of the existing systems require installation of additional software which could be an API or a runtime environment (e.g. Java). On the other hand, existing web-based visualization environments are directed to program visualization. Therefore, DAVE has the ambition to cover the need for a fully web-based algorithm visualization environment appropriate for introductory algorithmic courses at both, secondary and university, levels.

In our search for AV tools designed for algorithmic and introductory programming, we have used and evaluated many of the available AV educational environments. Although they provided valuable assistance, at both instructor and student levels, no one can cover the wide range of learning goals set by K-12 Computer Science (CS) curriculum. Taking into account the key principles for designing computer assisted learning environments and following a constructivist/inquiry pedagogical philosophy [30], our project has developed an AV environment which a) provides a live view of the code as it is executed; b) emphasizes and reveals the structure/logical steps of the algorithm under study, which are arguably the most critical or difficult for the students to grasp; and c) offers multiple representations allowing students to connect the logical view and the physical view of the algorithm (e.g. the way an array data structure is transformed during a sorting algorithm).

3.1. Context

In Greek upper secondary schools (Lyceums) Computer Science was introduced as a separate course since 1998. In the technological direction of Lyceum, an introductory course to procedural programming and algorithmic problem solving, entitled "Development of Applications in Programming Environments" (DAPE), is offered to third year (K-12) students. The existing National Curriculum suggests the use of a pseudocode, Pascal-like environment (in Greek), for the instruction of the basic algorithmic and programming structures. The students are introduced to procedural programming, variables, control and loop structures, arrays and sorting algorithms, as well as procedures and functions.

3.2. Key-design features

Following we present briefly the key-design features of DAVE:

Content: DAVE was designed to support instruction and students' learning activities in the context of DAPE course, according to the goals set by the curriculum. A wide range of algorithms with arrays are included in DAVE and supported by dynamic visualizations: bubble sort, insertion sort (in various versions), selection sort, quick sort, linear search, binary search, and merging of two sorted arrays. Figure 1 shows a screenshot of DAVE presenting the visualization of merging algorithm. This algorithm has a linear complexity and produces a sorted array that contains the elements of two given sorted arrays. The student can insert new input data to the source arrays or modify the pseudocode of the algorithm in order to investigate and reflect on its logic. Figure 2 illustrates the visualization of the binary search algorithm.

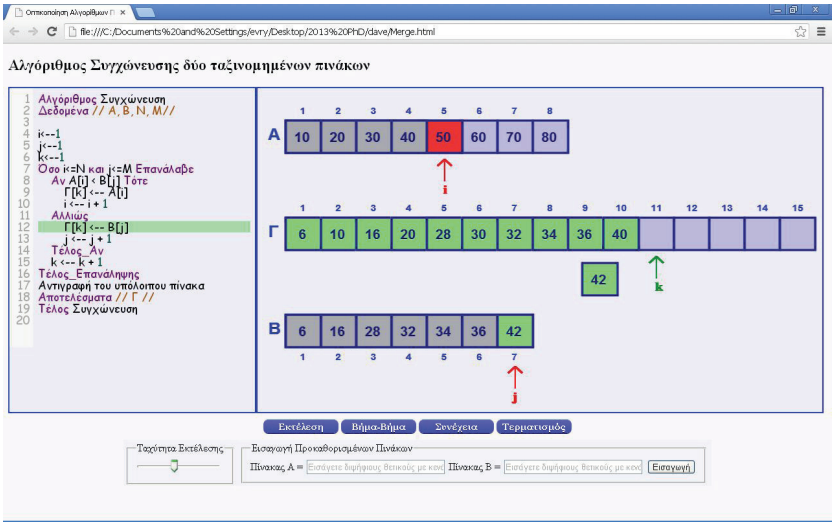

Fig. 1. DAVE interface: Visualization of the algorithm merging two sorted arrays.

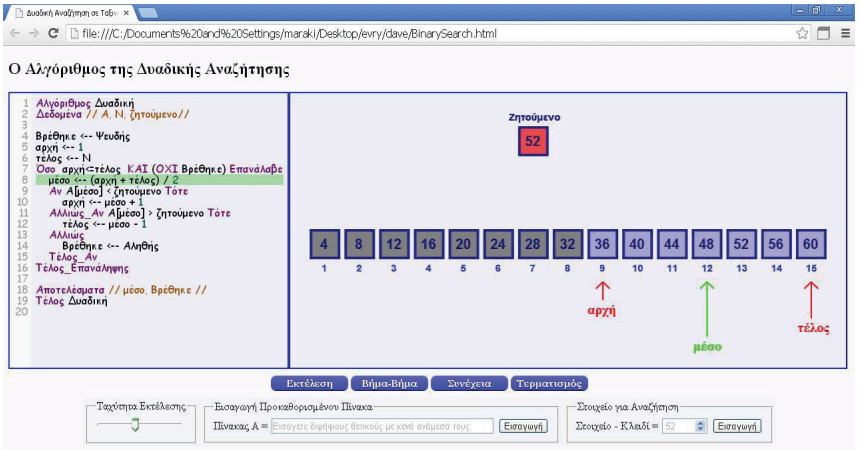

Fig. 2. Visualization of the binary search algorithm.

Visualization features: DAVE represents an array by a series of contiguous cells that can move during execution of the algorithm, in order to help students built efficient mental models and overcome their difficulties. The actions that are critical for the inherent logic of an algorithm are described as *interesting events* (e.g. swapping two array elements in sorting). Every algorithm has a different visualization in order to point out its interesting events. During algorithm execution, DAVE highlights the source code command corresponding to the current event. This helps students to link a programming command with the corresponding graphical presentation. Another important feature of the system is the animation of the control variables (i, j, k) in a loop command (e.g. FOR) which are visualized as arrows pointing to the respective array element. Control variables play also the role of array indices and, due to their automated change of value, they are considered as a source of difficulties for the students [6].
Interface and functionality: The user interface of DAVE consists of a browser window divided into three main components:

- The *source code editor* (left pane), that shows the algorithm that is visualized and highlights the line that is currently executed. The student is allowed to make modifications and restart the visualization.

- The *animation area* (right pane) where the animation of the algorithm is displayed, i.e. the constant changes of the graphical representations associated with the operation of the current algorithm command. The critical algorithmic parameters are highlighted or visualized by smooth transitions of objects.
- The *control panel* (down centre pane) includes operation buttons that support student-visualization interaction. There are four main control operations, e.g. start, pause, restart, and step-by-step execution. Input boxes help students to modify the input data of source arrays A, B in order to test algorithm execution for various data sets. The down-left slider aims to control the animation speed.

User control and engagement: The critical design principle for an effective algorithm visualization system is to promote students' engagement, not only during the execution of sample algorithms, but also through experimentation by modifying algorithm's code and input data. Using DAVE, the student can select the algorithm that he/she wants to visualize. During the execution of the algorithm, the system allows users to modify some parts of the code and, at the same time, to watch the corresponding visualization on the screen. The dynamic features above offer enhanced opportunities to the students to explore the behavior of the algorithm and to identify the critical logical steps and the structure of the algorithm.
Web features: Most web-based visualization systems are based on java applets. Therefore, users need to install the proper java runtime environment; following they download and launch the particular applet. Java applications are more heavy and demanding than scripts. DAVE is designed as a fully web-based system written in HTML5 and Javascript. Therefore, by simply visiting the hosting URL, it runs in any platform, operating system, browser or device. Educators and students do not need to install any additional software or download any special-purpose package. It is also fully client-based so there is no overload due to the communication with any server.

3.3. Architecture and implementation

There are three building components in DAVE (Figure 3): a) the *user interface* part, b) the *parser/compiler* and c) the *animation engine*. The user interface is actually a simple web page written in HTML5. The animation engine uses JavaScript and the HTML5 Canvas element. We have also used the CodeMirror [31] JavaScript component that provides a code editor in the browser and has a rich programming API and a CSS theme system which makes it highly customizable. The compiler unit of DAVE receives the source code as an input and produces an intermediate code which is executed by the animation engine. A predefined skeleton code for each algorithm is included in the animation engine. The parser/compiler scans the student's algorithm and fills the gaps in the code. This requires that the student has firstly selected the algorithm to be visualized from DAVE's list. Then he can do any modifications to the predefined code (e.g. a condition statement).

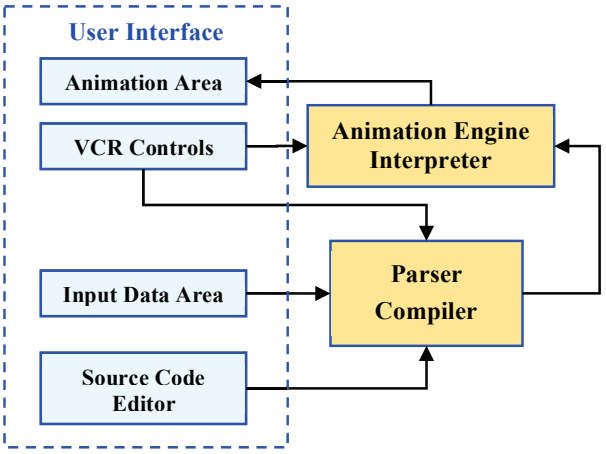

Fig. 3. The architecture of DAVE.

However, the student is not allowed to fundamentally change the nature of the algorithm. This system restriction was set because every algorithm has a unique animation; therefore, the animation engine should "know" in advance the critical parts of the algorithm to be animated. For example, compare the different visualization structure of the merging and binary searching algorithms, as shown in Figures 1 and 2.

DAVE is by design an extendable AV system. The prototype supports 2D smooth animations of array algorithms. It can easily visualize a new category (family) of algorithms just by adding a new visual data type, in the form of a JavaScript class, using methods which define the interesting events in student's code. In addition, the system can be extended and support other programming languages, like Pascal, C or Python. Therefore, the advantages of DAVE, comparing to other similar AV systems, can be summarized as following:

- It provides customized visualizations that depend on the nature of the given algorithm.
- It allows user to change input data in order to test algorithm behaviour in specific cases.
- It allows modification of the source code and step-by-step experimentation of the corresponding animation.
- It detects some categories of common students' errors, e.g. index value out of array bounds.
- It is easy to use and offers full platform compatibility due to the web-based architecture.

4. Empirical evaluation

4.1. Context and procedure

The study presented here was implemented in the context of DAPE course, which was briefly presented in the previous section, with the aim to evaluate a) students' perceptions of DAVE and b) students' performance on algorithmic problems with arrays. The study was implemented in two public high schools, at the end of the school year 2012-2013; it was expected that, at this period of time, the students were able to understand algorithms and write their own code involving arrays, conditional statements and loops.

A total of 45 students (17-18 years aged) were the primary sample group. The experiment was conducted during a two-hour session in the computer lab. The students were divided into four groups; thus they had the opportunity to work individually in their own computer. Before starting, they received a 10-minute tutorial/demonstration of the tool. Written instructions were also present on students' worksheet. The first activity was structured in separate steps in order to familiarize students with the tool and its functionality. The great majority of the students were able to use DAVE without the need for guidance by the instructor, in most learning activities. This is a strong indication of the usability of the tool.

4.2. The instrument

The research instrument was a written questionnaire including five learning tasks about algorithmic problems with arrays, designed specifically for this study. In order to respond to the tasks, the students had the opportunity to use DAVE and experiment with different algorithm versions, to execute a given code segment, to see the visualizations of the algorithms and modify their solutions. The tasks were mainly of two types:

- *explain code questions:* the students were given a code segment and they were asked to explain, in plain text, what this code does.
- *skeleton-code questions:* the students were given a code segment and they were asked to correct or modify it, in order to achieve an intended function or outcome.

In addition, the students were encouraged, but not obliged, to write their successive code versions and also their comments about DAVE during their learning tasks. After completing the investigation session, the students were asked to respond to the five questions in the questionnaire which aimed at recording their perceptions of the usability and usefulness of the environment. Finally, there were two open questions regarding students' evaluation comments of DAVE and their suggestions for improvements or new functionalities.

4.3. Students' perceptions of DAVE

Table 1 shows the results regarding students' views and perceptions of the usability and usefulness of DAVE. In overall, the students in the sample were very positive about the tool and the contribution of visualization towards understanding and solving algorithm problems with arrays.

Thirty out of 45 students gave additional comments regarding the usability and usefulness of DAVE and the visualizations included. Indicative students' written comments were as following:

"I had the chance to do something which helped me to understand how the sorting algorithm works"

"The detailed visualization by this software helped me to understand how the various operations are performed"

"I liked that, when I was modifying my algorithm, I could immediately see an intuitive visualization that helped me to discover and correct my errors".

Table 1. Students' perceptions of DAVE

Question	Positive (%)	Negative (%)
It was easy to me to understand how the visualization tool operates	89	11
It was easy to me to use the visualization tool	89	11
The visualization of looping counters (i,j) was helpful	78	22
The visualization helped me to understand sorting algorithms	96	4
Working with the visualization tool helped me to respond to algorithmic problems	98	2

Many students considered very important that they could access DAVE from their computer at home, due to the web-based architecture of the tool. Some students suggested also improvements and new features. Some of their suggestions have been included in the new version of the tool:

"I would like a backward button, to go back to the execution of the algorithm and repeat the difficult points"

"I would like the tool to support more algorithms, e.g. searching and two dimensional arrays".

4.4. Students' performance

Our findings showed that the majority of the students in the sample were actively engaged and devoted all the available time (two teaching hours of 45 min each) to respond to the learning tasks. Almost all students tried to respond to the questions by using DAVE, interacting with the visualization of a given algorithm, modifying the algorithm, testing their code and getting feedback from the algorithm visualization. The qualitative analysis of students' successive algorithm versions, extracted from the written worksheets, has given valuable information regarding students' algorithmic thinking. In the next, we present an indicative example of thinking evolution exhibited by a student during his work with a sorting algorithm through the visualization tool.

Task: *Input the following elements in an one dimension array (A): 32 38 98 54 60 90 20.*
Write a reverse bubble sort algorithm in order that, in every pass, the smallest element to be moved downward, as in Figure 4. For example, in the first iteration the smallest element (20) should be placed in the last position (A[7]). Write down your thinking and if possible your steps to reach the final answer.

	1st pass						
1	20	32	32	32	32	32	32
2	32	20	38	38	38	38	38
3	38	38	20	98	98	98	98
4	98	98	98	20	54	54	54
5	54	54	54	54	20	60	60
6	60	60	60	60	60	20	90
7	90	90	90	90	90	90	20

2nd pass					
32	38	38	38	38	38
38	32	98	98	98	98
98	98	32	54	54	54
54	54	54	32	60	60
60	60	60	60	32	90
90	90	90	90	90	32
20	20	20	20	20	20

Fig. 4. The first two passes of the requested sorting algorithm.

Following we present the first attempt of the student to develop a bubble sort algorithm that sorts numbers in descending order.

```
for i=2 to 7
    for j=7 to i step –1
        if □[ j ] > A[ j – 1 ]
then
        Swap( □[ j ], A[ j – 1 ] )
        end_if
    end_for
end_for
```

However, when this algorithm was executed in DAVE, the student noticed that this was not the algorithm requested. Despite that his algorithm gave the correct result on the screen (the array sorted in descending order) it has a different functionality than that described in the assignment. Next, the student modified his algorithm so that the smaller elements move down to the bottom instead of the larger ones moving to the top.

Now the functionality of this version (1) is closer to the algorithm requested. However the array is not sorted, since the result of the algorithm is the sequence 32 98 60 90 54 38 20. By using the visualization of this last version the student realized that the inner iteration should start from 2 and not from i (algorithm version 2).

The second version of the algorithm constitutes a correct solution of the given problem. After watching the visualization of the algorithm, the student noticed that there is no need to compare all the elements of the array in all the iterations. After extensive experimentation with DAVE, the student the student has modified the algorithm in its optimal/efficient form (version 3), which avoids unnecessary comparisons in order to achieve the expected sorting.

Algorithm Version 1	Algorithm Version 2	Algorithm Version 3

for i=2 **to** 7
  for j=i **to** 7
  if □[j] > A[j – 1] **then**
  Swap(□[j], A[j – 1])
  end_if
  end_for
end_for
	for i=2 **to** 7
	  for j=2 **to** 7
	  if □[j] > A[j – 1] **then**
	  Swap(□[j], A[j – 1])
	  end_if
	  end_for
	end_for
		for i=2 **to** 7
		  for j=2 **to** 9–i
		  if □[j] > A[j – 1] **then**
		  Swap(□[j], A[j – 1])
		  end_if
		  end_for
		end_for

According to the researcher's notes during the investigation sessions, the visualization of loop counters was an important feature of DAVE which enhanced students' performance. Loop counters play also the role of array indices. The representation of indices as arrows that point to the corresponding array elements was also helpful. In addition, an arrow points to the next array element when the corresponding index is increased. Thus, when the students tried to increase an index pointing to the last array element, the arrow will move out of the right boundary of the array, and an error dialog was displayed on the screen indicating access out of array boundaries. This is an indicative example of the way some students were supported by DAVE to detect their logical errors. In addition, DAVE helped the students in the sample to overcome a well-known difficulty in understanding the automated value increase of the loop control variables (array indices) in FOR command [6].

5. Summary and future work

This paper presented the design framework, the architecture and the pedagogical features of DAVE, a new web-based dynamic algorithm visualization environment aiming to enhance secondary education students' learning about sorting algorithms in arrays. Currently, the prototype supports the pseudo-code programming language which is used in Greek upper secondary schools, in the context of the introductory course about algorithms and programming. However, DAVE can be extended by adding other languages (such as Pascal, C, Python) and other data structures (such as graphs, trees, stacks and queues). Therefore, it can support programming courses at university level a well.

From the learners' perspective, DAVE may offer to introductory courses about algorithms not only by facilitating constructivist and discovery learning activities but also by supporting different types of learners. The

empirical evaluation of the system, implemented in two public high schools, gave promising results regarding K-12 students' perceptions of DAVE and their performance on algorithmic problems with arrays. The visualizations with DAVE helped them to identify their logical errors and modify their code to solve the particular problems assigned.

The preliminary research data showed that, in most cases, DAVE promoted students' engagement and experimentation with algorithm visualizations; there were strong indications that DAVE supported students' algorithmic thinking and helped them to overcome learning difficulties about sorting algorithms. In addition, students were positive about the environment, its usability and its usefulness to support their learning. The analysis of research data is in progress and we expect to reveal valuable information about students' possible patterns of developing algorithmic knowledge and solving problems with arrays.

References

1. European Commission. *Digital literacy report: A review for the i2010 e-inclusion initiative*. Commission Staff Working Document; 2008.
2. Dale NB. Most difficult topics in CS1: results of an online survey of educators. *SIGCSE Bull.* 2006;**38**(2): 49-53.
3. Robins A, Rountree J, Rountree N. Learning and teaching programming: A review and discussion, *Computer Science Education* 2003;**13**(2): 137-172.
4. Soloway E., Spohrer, JC. *Studying the novice programmer*. NJ: Lawrence Erlbaum; 1989.
5. Jimoyiannis A. Using SOLO taxonomy to explore students' mental models of the programming variable and the assignment statement. *Themes in Science and Technology Education* 2011;**4**(2):53-74.
6. Danielsiek H. Detecting and understanding student's misconceptions related to algorithms and data structures. *Proceedings of the SIGCSE 2012 Technical Symposium on Computer Science Education*. Raleigh, North Carolina: ACM Press; 2012. p. 197-201.
7. McGettrick A, Boyle R, Ibbett R, Lloyd J, Lovegrove L, Mander K. Grand challenges in computing education – A summary. *The Computer Journal* 2005;**48**(1):42-48.
8. Rößling G. The Animal algorithm animation tool. *5th Annual SIGCSE/SGCUE Conference on Innovation and Technology in Computer Science Education*, ITiCSE'00. Helsinki, Finland; 2000. p.37-40.
9. Pierson W, Rodger S. Web-based animation of data structures using JAWAA. 29th *SIGCSE Technical Symposium on Computer Science Education*. 1998. p. 267-271.
10. Moreno A, Joy MS. Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science* 2007;**178**:51-59.
11. Naps TL. JHAVE: supporting algorithm visualization, *Computer Graphics and Applications* 2005;**25**(5):49-55.
12. Korhonen A, Helminen J, Karavirta V, Seppala O. TRAKLA2. In: Pears A, Schulte C. editors. The *9th Koli Calling International Conference on Computing Education Research*. Koli Calling '09. 2009. p. 43-46.
13. Hundhausen D, Brown J. What you see is what you code: A 'live' algorithm development and visualization environment for novice learners. *Journal of Visual Languages and Computing* 2007;**18**(1):22-47.
14. AlgoViz.org Bibliography. Annotated bibliography of AV literature. http://algoviz.org/biblio, 2011.
15. Ma L, Ferguson J, Roper M, Wood M. Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education* 2011;**21**(1):57-80.
16. Guo JP. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In: *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education*. New York: ACM. 2012. p. 579-584.
17. Vrachnos E, Jimoyiannis A. Dave: A Dynamic Algorithm Visualization Environment for novice learners. *8th IEEE International Conference on Advanced Learning Technologies* 2008. p. 319-323.
18. Garner S, Haden P, Robins A. My program is correct but it doesn't run: a preliminary investigation of novice programmer's problems. In: *ACE'05: Proceedings of the 7th Australasian conference on Computing education*. 2005. p. 173-180.
19. Du Boulay B. Some difficulties of learning to program, In: Soloway E, Spohrer JC, editors. *Studying the Novice Programmer*, Hillsdale, NJ: Lawrence Erlbaum Associates; 1986. p. 238-299.
20. Helminen J, Malmi L. Jype – a program visualization and programming exercise tool for python. In: *Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10*. New York: ACM Press; 2010. p. 153-162.
21. Sorva J, Sirkiä T. UUhistle: a software tool for visual program simulation. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. New York: ACM Press; 2010. p. 49-54.
22. Fouh E, Akbar M, Shaffer C. The role of visualization in computer science education. *Computers in the Schools* 2012;**29**:95-117.
23. Baecker R. Sorting out Sorting. Narrated colour videotape, 30 minutes, *Presented at ACM SIGGRAPH'81*, 1981.
24. Stasko JT. Using student-built algorithm animations as learning aids, In: *28th SIGCSE Technical Symposium on Computer Science Education*; 1997. p. 25-29.
25. Karavirta V, Korhonen A, Malmi L, Stålnacke K. MatrixPro: A tool for on-the-fly demonstration of data structures and algorithms. In: *Proceedings of the Third Program Visualization Workshop;* 2004. p. 26-33.
26. Rajala T, Laakso M-J, Kaila E, Salakoski T. Effectiveness of program visualization: a case study with the ViLLE tool. *Journal of Information Technology Education* 2008;**7**:403-407.

27. Hundhausen D, Douglas S, Stasko J. A metastudy of algorithm visualization effectiveness. *Journal of Visual Languages & Computing* 2002;**3**(3):259-290.
28. Kehoe C, Stasko J, Taylor A. Rethinking the evaluation of algorithm animations as learning aids: an observational study, *International Journal of Human Computer Studies* 2001;**54**:265-284.
29. Byrne D, Catrambone R, Stasko J. Evaluating animations as student aids in learning computer algorithms. *Computers & Education* 1999;**33**(4):253–278.
30. Jimoyiannis A. Computer simulations and scientific knowledge construction, In: Cartelli A, Palma M, editors. *Encyclopedia of Information Communication Technology*. Hershey, PA: IGI Global; 2008. p. 106-120.
31. CodeMirror, http://codemirror.net, 2013.