System for automatic generation of algorithm visualizations based on pseudocode interpretation

ABSTRACT
Algorithm visualization systems have not been as widely adopted by computer science educators as it might be expected from the firm belief that they can enhance computer science education. Two key impediments for widely adopting AV technology in mainstream computer science are: effectiveness and enhancements of learning with visualization and effort needed to create algorithm visualizations. In this paper, we present the interpretation based system capable of automatic creation of algorithm visualizations by interpreting unmodified algorithms written in pseudocode. Although system is interpreting unmodified source code (code without any annotations for triggering appropriate visualization routines), due to the ability to automatically detect interesting events system is able to create visualizations at a sufficiently high level of abstraction so that the emphasis is on algorithm conceptually relevant principles. Providing users with full control over input data set and by accompanying animation with explanatory messages, highlighting currently executing pseudocode line and providing possibility to inspect variable values at any step visualizations created by our system that can enhance learning and help students mastering algorithms basic concepts.

Categories and Subject Descriptors
H.5.1. [Information interfaces and presentation]: Multimedia
Information Systems – Animations.

General Terms

Keywords
Algorithm visualization, code interpretation, pseudocode, automatic animation generation, automatic interesting event detection.

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

Conference '10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010 …$15.00.

1. INTRODUCTION
Understanding of computer algorithms is itself a very difficult task which becomes even more difficult when it comes to real programming language implementation when student’s attention from algorithmic conceptually relevant principles gets distracted by syntax and semantic details specific to programming language. Algorithm visualizations (hereafter referred to as “algorithm visualizations” or “AVs”) are used in computer science education since the early eighties. By graphically representing computer algorithms, their various states and animating the transitions between them, algorithm visualization technology aims to help computer science students understand how algorithms work. There are various surveys on using visualization as an aid for computer science education. Although AV’s are naturally attractive to educators, who nearly universally view them positively, consistently “liked” by students and despite its intuitive appeal as a pedagogical aid, AV technology has failed to catch on in mainstream computer science education. Two key impediments for widely adopting AV technology in mainstream computer science are: effectiveness and enhancements of learning with visualization and effort needed to create algorithm visualizations. Even though there is a firm belief that graphical representations of algorithms are learning aids, some initial studies designed to substantiate the educational effectiveness of AV’s have shown that using visualization is not advantage at all or an advantage that can be only partially attributed to AV technology. However, later studies (e.g., [21]) have shown that AV technology could enhance and improve learning if visualization is interactive enough. In order to be educationally effective system must support student interaction and active learning (support for changing input data sets, making predictions regarding future visualization states, programming the target algorithm, supplement the visualization with appropriate text such as synchronized pseudocode and other descriptions of the algorithm). Besides effectiveness and enhancements of learning with visualization the wide adoption of the visualization system depends on one more important factor – effort needed to create algorithm visualization. Price et al. [22] define four different roles in algorithm animation: user, visualizer, software visualization software developer (or simply developer) and programmer. The user views and interacts with an animation specified by the visualizer. The underlying animation system is designed and implemented by developers. Finally, the programmer is the implementer of the visualized algorithm. Each role has different expectations of algorithm...
In general, majority of those findings pointed to two key obstacles to visualization technology’s widespread adoption:

- From the learner’s perspective, the visualization technology may not be educationally beneficial. More interaction, than just viewing animations, is needed to obtain learning improvements.
- From the instructor’s perspective, the visualization technology may simply incur too much overhead to make it worthwhile.

2. MOTIVATION

Motivation for the development of this system was twofold — two different problems were addressed. First driving force behind our motivation - Ministry of science, education and sport in Croatia has introduced official specification for pseudocode [1] with allowed statements, operators and functions as well as 1:1 mapping to conventional programming languages. With this specification pseudocode became the official language in programming exercises for high school, state graduation exam, and entrance exam for most colleges as well as for national competition in informatics for primary and high school students. Despite this fact, as far as we know, no one provided parser or interpreter for such specified pseudocode nor anyone developed environment for learning pseudocode (kind of integrated development environment) in which students would be able to test their programming skills. As second motivation driver we had heterogeneous group of freshmen students attending “Artificial Intelligence” course. They had finished very different high schools and their general programming skills varied a lot (some of them were not familiar with any conventional programming language at all). What they did have in common was – they all had prior knowledge about pseudocode. Within the course they were expected to master some basic algorithms related to artificial intelligence like breath first search, depth first search, limited depth first search and a like. Although most of the students passed the course exam we have noticed that they were missing this conceptual link / connection between pseudocode and algorithm behavior. They were able to write algorithm pseudocode and to explain algorithm behavior but they weren’t able to connect line of pseudocode with specific algorithm routine. In other words, most of them were not able to answer questions like - how it would reflect on algorithm behavior if this line of code were different?

At that point we realized that it would be of great help if we could develop a system that will be able to visualize appropriate algorithms purely by interpreting pseudocode.

3. RELATED WORK

Algorithm visualization history can be divided in two phases [2] – systems that came before the rise of Sun’s Java programming language and wide content sharing on Internet and system that came after. Systems developed in first phase mostly came packaged with pre-generated AV’s but allowed educators and students the freedom to implement other AV’s using special purpose scripting languages or by annotating real programming language. Since widespread use of Java, systems have moved away from authoring toolkits towards suites of “canned” AV’s and they are mostly distributed as operating system and environment independent collections of AV’s. Balsa (Brown Algorithm Simulator and Animator) [3] is considered as the first major interactive software visualization system. Balsa was used extensively in the Brown University Workstation Project as an educational tool for teach fundamental computer science concepts using program-generated animations. Another recognized system of the early years of algorithm animation is Tango (Transition-based Animation Generation) [4] developed by John Stasko at Georgia Tech. Tango was one of the first systems supporting smooth animation. In subsequent years number of animation systems has risen steadily. These systems were much different from each other according to certain characteristics and implemented features. Taxonomy of software visualization [5] introduced the categorization by which systems were categorized by: visualization specification style (animation creation approach), visualization specification technique (specifying connection between visualization and algorithm source code) and animation creation effortlessness (how much time and effort is needed to create algorithm animation). In original taxonomy visualization specification style was measured using terms like hand-coded, library, and automatic. Since that taxonomy is bit outdated alternative categorization was introduced in [6] and refined in [7]:

- Topic-Specific Animation - animations built specifically for some topic and often they are coming in form of stand-alone animations instead of algorithm animation systems.
- Direct Manipulation - In direct manipulation [8], the animation is specified by manipulating graphical objects. Good example of AV systems using direct manipulation to generate animations is MatrixPro [9].
- API-based Generation - API-based generation involves method invocations using a special visualization application programmer’s interface (API). Perhaps most popular system using this approach is JHAVE [10].
- Scripting-based Generation - animations are described using some intermediate format, usually a textual format. The expressiveness of those scripting-like languages for describing animation is strictly limited to animation purposes. Methods, variables, loops and such stuff are usually not supported. Good example of such system is Animal [11].
- Declarative Visualization - visualization is specified by defining mapping from a given program state to graphical representations. It uses abstract mathematical expressions which can result in complex visual representations. A good example of this approach is the ALPHA language [13].
- Code Interpretation - Some animation systems offer a direct visualization of algorithms from the underlying code. This could be achieved by relying on debugger used to retrieve the current state, by preprocessing the underlying source code or by interpreting source code “as is”. Example of system using code interpretation for animation generation is Jeliot 3 [14] – used to visualize execution of Java programs. Main strength of code interpretation-based animation systems are effortlessness and tight connection between the source code...
and its visualization. However, code interpretation approach has some weaknesses also.

While creating algorithm visualization one of most important tasks is to specify connection between visualization and algorithm source code. By visualization specific techniques systems can be categorized as follows [15]:

- **Event driven approach** consists of identifying interesting events and annotating the algorithmic code with calls to suitable visualization routines. The interesting event approach was pioneered by Balsa [3] and has been used in many algorithm animation systems including Tango [4].

- **State driven approach** consists of specifying mapping between program and visualization states, usually constructed by the visualizer before program execution. Good example of this approach is its early adopter Pavane [16].

- **Visual programming** makes programs easier to specify by letting users create programs manipulating program elements graphically rather than by specifying them textually. Example of this approach can be found in Dance [17].

- **Automatic Animation** – although being simplest way to specify an animation from the algorithm developer point of view, automatic algorithm animation is extremely difficult task. Perhaps the most popular system which is producing animation automatically (from source code) is Jeliot [14]. This approach was more often used for visualizing program execution rather than visualizing algorithms because algorithm visualization requires animation on much higher level of abstraction (more conceptual level) which is extremely hard to achieve with automatic animation approach.

As it’s already pointed out - time and effort needed to learn and create algorithm visualizations are one of the key reasons why AV technology is still not widely used by computer science educators. Animation creation effortlessness is subjective matter and includes many factors. One characteristic that has a great influence on the effort required is the ability to use the system on-the-fly. With possibility to create animation on-the-fly lecturer is able to show different use cases, to answer to students “what-if” questions and to show same animation multiple times with different input sets which is not possible with predefined animations.

### 4. DEVELOPED SYSTEM

In order to make algorithm animation creation process as effortless as possible we decided to make our system interpretation based in a way that it will be able to produce animations automatically just by interpreting pseudocode “as is”. Introducing the approach of automatic interesting events detection we managed to avoid annotating the source code and by interpreting unmodified source code still end up with visualization at a sufficiently high level of abstraction to be educationally effective for students. For the sake of simplicity we have decided to make first version of the system course-specific and start with smaller set of search algorithms performing on tree (depth first search, breadth first search, depth limited search, etc.).

#### 4.1 Pseudocode

As already mentioned, ministry of science, education and sport in Croatia has introduced official specification for pseudocode [1] with allowed statements, operators and functions as well as 1:1 mapping to conventional programming languages. This fact gave us the opportunity to develop a system in which students will be able to write an algorithm in pseudocode and see its visualization without the need for prior knowledge of any conventional programming languages. Writing algorithms in pseudocode allows students to concentrate on algorithm basic concepts rather than thinking about algorithm implementation in specific programming language. To meet the needs of this system we have extended this pseudocode specification by adding definition for functions, function calls, arrays, structures, few new keywords, few built-in functions (like print – to print something to the console), etc. For such specified pseudocode we have developed parser and interpreter. We have also leveraged the fact that pseudocode specification includes 1:1 mapping to some

![Figure 1. Developed system user interface (system in animation mode)](image)
conventional languages (Pascal, C/C++, Python) and as proof of concept we have developed converter that is capable of converting pseudocode to Python language. We chose Python because it leaves us the possibility to embed Online Python Tutor (Embeddable Web-Based visualization of Python programs) [20] in our system and thus make our system capable of visualizing program execution.

4.2 System design

System has two modes of operation – edit and animation mode. While in edit mode, system allows user to write/edit algorithm and to create structure on which algorithm will perform. Once that is done, by clicking button run system goes to animation mode. In animation mode user is allowed to navigate through algorithm animation while most other components of the user interface become disabled (read-only). User interface of the system in animation mode is shown in Figure 1.

In the part of the figure numbered 1 there is code editor for pseudocode with some useful features like live pseudocode syntax check, syntax highlighting, code folding, etc. When system is in edit mode, in code editor users can write their own program/algorithms or choose one of the available algorithms. At any time, users can switch to Python tab and see their pseudocode converted to Python programming language. Python code is read-only, but syntax is still highlighted. When system is in animation mode code editor becomes read-only and reflects code execution in a way that it highlights the line that is currently being executed.

Part of the figure numbered 2 contains visualization area. When system is in edit mode, visualization area is used to create tree structure on which selected algorithm will perform. The root node is created automatically with default value set to “…”. When we point the cursor over the node, node value becomes editable and it can be easily changed just by entering new value. Also, we can easily add child to the node just by double clicking on it. All those actions (add child, remove child, etc.) are also available through the context menu when we right click the node. Context menu contains two additional options – “Mark as source node” and “Mark as target node” as an input to the search algorithm – which node we are searching for and where to start. There is also possibility to create random tree (with random number of nodes and random values) just by clicking “Generate random tree” button. Once the system is in animation mode it’s no longer possible to change the tree structure. At this point navigation bar becomes enabled and we can navigate through animation step by step by clicking forward and backward buttons or we can achieve smooth navigation by clicking autoplay button. At any time smooth animation can paused and we can continue navigation using forward and backward buttons. There is also a slider for faster navigation through animation steps. In the top right corner of this section there is a label named “how to create a tree?” when system is in edit mode and “how to use animation?” when system is in animation mode. By pointing cursor over that label we will get detailed instructions of how to create a tree or how to use an animation.

In the part of figure numbered 3 there is a console containing program / algorithm output (for example messages outputted with print built-in function) and animation accompanying explanatory messages (for example “comparing two nodes”). Besides output, there is also an error tab containing all potential errors from parsing and interpreting written pseudocode.

In the part of figure numbered 4 we can see execution stack and observe variable values at any point of algorithm animation or program execution. Variable values, as well as the highlighted line of code in code editor, are synchronized with animation and changes accordingly by navigating through animation using navigation bar.

There is also a help section containing detailed description of provided algorithms, tutorial for system usage, specification of pseudocode extension and a like. Help section is shown on the Figure 2.

4.3 Automatic detection of interesting events

We are pairing each node in the tree with accompanying object in memory which we have called animation object. While interpreting pseudocode our interpreter detects every access to animation object and then, if access is characterized as interesting event, some visualization routines are triggered. Regarding algorithms for searching within the tree structure, accesses that we have characterized as interesting events are: accessing animation object (reading value property of animation object), comparing values of two animation objects, assigning value to animation object, pushing animation object to queue structure (array) and removing animation object from queue structure. For some other types of algorithms different interesting events could be specified. For example if we want to visualize some sorting algorithm then swapping two elements should be detected and characterized as interesting event. Within this work we have considered only the algorithms for tree searching, tree as the only structure that algorithms are performed on and tree node as the only type of animated object. If we ever decide to extend this system to support visualization of other type of algorithms (for example sorting algorithms) then we should introduce array as new structure that sorting algorithms will perform on. We will have to introduce array member as new type of animation object and to specify which events should be characterized as interesting events when sorting algorithm is performing on array. This process could be automated to some extent in the way that the system itself determines which events to consider as interesting events depending on which structure algorithm is performing on (for example if algorithm is performing on array than system can conclude that comparing value of two members or swapping position of two members
should be considered as an interesting event). On the other hand, we can let the user to decide which events are interesting events in context of specific algorithm animation.

4.4 Visualization generation
Regarding visualization specification style developed system is using code interpretation approach. The vast majority of other systems that use this approach annotates source code in a way that they injects calls to visualization routines in places that are considered interesting events to achieve visualization at a sufficiently high level of abstraction. Thanks to ability to automatically recognize interesting events our system is capable of producing algorithm animation automatically and visualizing algorithms on high level of abstraction by interpreting unmodified source code. Using code interpretation approach makes visualization generation as effortless as possible and provides tight connection between the source code and its visualization. Every change in source code is automatically reflected on algorithm visualization which improves user understanding of how algorithm works. Animation is created in parallel with the interpretation of the source code and when the interpreter has finished animation is ready and divided into a certain number of steps. Animation step is created when interesting event is recognized and after any line of source code.

For creating algorithm animations we have used JSAV (The JavaScript Algorithm Visualization Library) [19]. By accompanying animation with appropriate explanatory messages and updating of variable values at every step we intended to improve user’s understanding of the algorithm.

5. EVALUATION OF THE SYSTEM
We have conducted preliminary evaluation of our system on the students taking the course of artificial intelligence. The students (n=23) had short introduction to system during which we showed them how they can solve the problem of missionaries and cannibals using our system. After this short introduction, students were asked to solve travelling salesman problem in a similar way. 13 out of 23 of them solved the task correctly and web-based survey “User Interface Usability Evaluation with Web-Based Questionnaire” [23] that they have filled out upon the completion of the exam shows that they are general quite happy with the system and believe that system will help them in understanding the algorithms. Original survey contains 12 questions from which we have extracted 7. Results of the conducted survey are shown in Figure 3. As future work, we plan to conduct a long term evaluation of this approach.

6. CONCLUSIONS AND FUTURE WORK
Our primary goal was to develop an algorithm animation system which will be able to animate algorithm execution purely by interpreting pseudocode at a sufficiently high level of abstraction to be useful for students in understanding of the algorithms. We believe the system that we have developed could be considered as more than just system for algorithm visualization. Within the scope of this work we managed to achieve following goals:

- We have extended formal pseudocode specification, parser and interpreter for such specified pseudocode as well as converter which is able to convert pseudocode to Python.
- Our system is web based, education oriented, cross-browser compatible (platform independent)
- Since animation creation process is automatic in our system, it’s truly effortless for both - lecturers and students. They can be considered as system consumers because visualizer role is avoided since the system is interpreting code “as is”.
- By accompanying animation with explanatory messages, highlighting currently executing pseudocode line and providing possibility to inspect variable values at any step our system is capable of producing visualization that hopefully will enhance learning and help students mastering algorithms basic concepts.
- Our system can be also used as kind of integrated development environment for anyone who wants to test their skills and learn to program in pseudocode. In editor supporting features like pseudocode live syntax check, syntax highlighting, code folding, etc. users are able to write their programs, execute them, see console output and in debugging manner inspect variable values, see execution stack and observe code execution line by line.
- Thanks to the possibility of converting pseudocode to Python, with relatively little effort it would be possible to embed Online Python Tutor [20] into our system and see visualization of program execution which would make our system fully featured program execution visualization system and visual debugger.
Our system contains comprehensive documentation with detailed description about included algorithms, pseudocode extension, system usage tutorial and a like. So far, our system contains only front-end component and doesn’t require internet connection to work properly. One of the plans for the future development is introducing back-end component with database after which we will be able to implement features like: Let user to write and save his own algorithm, save created structures on which algorithm performs (just trees so far), exercises with automatic assessment, feedback form to measure the level of users satisfaction, etc. Regarding scope, at this point developed system can be considered as course-specific system (supports visualization of searching algorithms from artificial intelligence). As future work, we have a plan to extend its application area to domain specific (to allow visualization of other types of algorithms like sorting algorithms, etc…). Also, as next step we intend to provide the converters which will be able to convert pseudo code to C/C++ and Pascal.

7. REFERENCES


