Algovision or how to animate algorithms

Luděk Kučera
Charles University, Prague, Czech Republic
ludek@kam.mff.cuni.cz

Abstract
The paper describes the design philosophy of Algovision, a collection of algorithm visualization applets that was developed at Charles University as a support for the Algorithm and Data Structures course. It is argued that a successful algorithm visualization applet should be a complex system that helps a student to understand *why* the applet works, i.e. to understand an algorithmic idea that is behind.. In certain cases we even use animations of mathematical proofs. Many other features must also be provided, explaining, e.g., implementation issues, applications etc

Keywords: Algorithm animation, algorithm verification, algorithm teaching

1. Introduction

Algorithm animation is a term that denotes a visual presentation of algorithms using images that change in time.

An algorithm is a sequence of instructions that tells a human or a computing device what specific steps to perform (in what specific order) in order to carry out a specified task. In computer science, algorithm usually operates on mathematical objects like matrices, graphs, geometrical bodies etc.

Therefore, when describing, teaching or learning an algorithm, we have first specify objects that the algorithm manipulates (that translate into variables or data structures of the corresponding computer code) and then a sequence of operations that is performed on the objects (that translate into instructions of the code).

Recent information and communication technologies provide tools for presenting information in a visual form that changes in time and as a response of actions of a viewer or user of the system. It is not surprising that many teachers of mathematics and computer science immediately recognized potential of ICT in algorithm teaching/learning. It seems that it is sufficient to choose a visual representation of data structures and to show how they change in time during computation.

Standard ways of static visual representation of mathematical objects can be found in textbooks, e.g., matrices as square tables of numbers, graphs as a collection of circles representing nodes that are evenly placed in an area of the plane and are connected by lines representing edges, geometric bodies by their photography-like projection into plane etc.

Evolution of data structures is usually done either by a discontinuous replacement of the previous value, form or shape by a new one or by some kind of a continuous transformation over certain time interval.

Even though algorithm animation can be traced to earlier times (the used technology being TV or a classical movie, see e.g. [Ba81]), a true advent of algorithm animation occurred in 90's when personal computers and graphical terminals became widely available. A large number of systems have been developed, e.g. BALSA, ZEUS, Tango, Polka, Samba, for an extensive overview see [KS02] and [SDBP]. An even larger number of ad-hoc animations can be found at the web.

However, even during 90's it had been recognized that a use of algorithm animation in algorithm teaching is far behind our original expectations and this state persists until now. E.g., J. Stasko, one of the leading experts in algorithm animation, said in 1996 ([KS96]) that "A number of studies have

found that using animation for explaining dynamic systems had less beneficial effects on learning than hoped. Those results come as a surprise to many instructors and students in computer science where animation is becoming an increasingly popular tool for teaching algorithms."

Several approaches were tried to make a use of algorithm animation more frequent. One direction of development was based on the assumption that use of algorithm animation is restricted because writing a good animation requires enormous effort and aimed to development of systems for transforming computer programs into animations automatically. Another researchers and teachers believed that student themselves should write animations using systems supporting such activity. All approaches that were oriented to systems for animation writing rather than to animations were successful only partially. We will discuss this issue later.

2. Animal or anima?

When speaking about algorithm animation, many people believe that the term "animation" comes from the word "animal" – a living being that moves, follows sometimes quite complicated but fixed patterns of behavior but it doesn't posses higher intellectual abilities. Indeed, this is supported by many existing algorithm animations that move, follow sometimes quite complicated hidden scheme, but a viewer, unless he or she understands the animated algorithm well, is not able to figure out what is going on the screen or display.

However, the true root of "animation" is a latin word "anima" that means "soul". A problem of a typical algorithm animation is that it lacks soul or, better to say, it does not aim to reveal the soul of the algorithm. By this, we mean an algorithmic idea that is behind the computational process.

Creative scientists have intuition; they "see" the principle of an algorithm in a way that is difficult to express and which imply not only *how* the algorithm works, but also *why* it works. (This, of course, is not limited to algorithms, but is common to all sciences.) Many of them admit that intuition manifests in form of images that appear in their mind, and it would be sufficient to project the images to display.

It is obvious that the last sentence of the previous paragraph is just a metaphor. Like *tao*, an intuition that can be spoken is not the true intuition; "spoken" should be understood in the broadest sense to involve an explicit visual representation. "Images" in mind are vague and they definitely can't be grasped; perhaps they are not images at all.

Fortunately, there is a way to define a notion of an "idea behind an algorithm" in a strict mathematical sense. The formalism comes from a theory of program verification. An input for a program verifier is a program plus two logical formulae. The goal is to prove that the program applied to input data verifying the first formula stops after finite number of steps and gives an output that verifies the second formula.

A method to prove halting is to find a function that maps any possible collection of values of variables of the program into an element of a well-ordered set (e.g., non-negative integers) and to prove that the value of the function decreases during *each* step of the computation. The initial value of the function gives usually information about the time complexity of the program.

A method to prove correctness of the output (i.e., that the output verifies the second formula) is to find a third logical formula, called an *invariant*, such that (i) the input formula implies the invariant, (ii) the invariant together with conditions that are fulfilled when the program terminates imply the output formula, and (iii) during any elementary step of the computation the validity of the invariant is preserved.

Once an invariant and a termination function are given, it is usually quite simple (though often very tedious) to prove that the termination function strictly decreases and the invariant remains satisfied. The main problem is, however, how to find a proper invariant and a termination function. Using

unsolvability of halting problem, it can be inferred that an invariant and a termination function can't be found algorithmically.

When asked by students how to find an invariant for a given program, I usually answer that this task is in fact quite simple. One has to *understand* what the program is doing and then it is rather straightforward to figure out logical constraints that are verified by the values of the variables, and it is sufficient to write them down.

Conversely, knowledge of an invariant tells a user what is the goal of the computation. E.g., the Dijkstra's shortest path algorithm marks certain nodes as *definitive*; at the beginning only the source is definitive and the algorithm stops when *all* nodes are definitive. The termination function is the number of nodes that are not definitive, and the invariant says that the estimation of the cost of a node *u* (a variable that is maintained by the algorithm) is always the length of the shortest path from the source into *u*. The easily understandable termination function and invariant determine uniquely the algorithm: we need a loop that increases the number of definitive nodes by 1 during each iteration. Since claiming a node as definitive might violate validity of invariant, the loop body involves updating variables in a way that restores the invariant. In this way, understanding the Dijkstra's algorithm means to know that the definitive part of the graph is always in a required state while evolving from the initial trivial form to eventually absorb whole graph, i.e. to know the invariant.

Since our goal is to create animations that explain *why* the algorithm works instead of fabricating movies that show *what* happens, we try to design visual representation in a way that makes it easy to see and infer the invariant and/or the termination function. Very often this requires using a new way of presenting the data.

The previous analysis also explains why the two approaches mention above, namely automated building of animation from a program and animations created by student do not work as it has been expected.

It is not possible to infer the invariant from a code in an automated way, and therefore the method could only produce animations "without soul" that have only limited value (this issue will be illustrated in the next section when discussing algorithms of Dijkstra and Bellman-Ford).

In order to write a good animation, a student should have a prior understanding of the method. This, however, reverses the time sequence, since we wanted that a student develops understanding as a result of writing an animation.

3. Algovision

Algovision is a collection of applets that visualize algorithms from different fields of Computer Science. Presently the system covers data structures (lists, trees, heaps), sorting (Mergesort, Quicksort, Heapsort, Bubblesort), graph algorithms (searching and components, shortest path, minimum spanning trees, flows in networks – both path augmenting and preflow push), arithmetic algorithms (carry look-ahead addition and FFT), geometric algorithms (convex hull and Voronoi diagram in 2D) and linear optimization (simplex algorithm in 2D and 3D). The collection essentially covers a 2 semester course on algorithms and data structures for undergraduate computer science students at Charles University.

Algovision philosophy is the one described in the previous section. The system is not an implementation of previous theoretical issues; conversely, our philosophy developed together with the development of the system that was originally conceived as a collection of standard "animal" animations. As Algovision was used in the class, it soon became apparent that a teacher needs much more features than a simple-minded animation, which even turned out to be almost useless. Therefore the applets were extended to cover at least partially the other parts of the lecture, namely

the analysis of correctness, termination and computational complexity, and the previous section is a conclusion of the development process.

Now the Algovision philosophy will be illustrated on several applets. Let us start with two shortest path algorithms – Dijkstra and Bellman-Ford. Both of them can be viewed as an implementation of a single general labeling scheme; the only difference being that Dijkstra's algorithm uses a priority queue (among all reached but unprocessed nodes, it chooses the one with the smaller cost estimation), while the algorithm of Bellman and Ford uses a FIFO queue (the oldest node in the queue is selected). This would suggest that the same animation idea is used in both cases, and this is really the case for most of the available shortest path animations (and for the Algovision applets in the "what happens" mode).

However, there is striking difference in behavior of these two algorithms that is also reflected by completely different invariants used to prove termination and correctness.

The Dijkstra's algorithm applet in Algovision, when in the "why" mode, redraws the input graph in such a way that the *x*-coordinate of any node is proportional to the cost estimation of the node. Hence, as the cost estimation decreases during the computation, a node moves left. Positions of nodes make it easy to understand how the computation works.

When computing using Bellman-Ford algorithm, nodes receive the final value of the cost estimation (equal to their cost) in an order that is the same as a BFS order in the tree of shortest paths. Thus, the idea behind the correctness and termination proof is best explained when the computation is first animated in the standard way using the original placement of nodes, the tree of shortest paths is shown, nodes are reshuffled so that their *x*-coordinate is proportional their depth in the tree and the computation is performed again. During the repeated computation, receiving the final cost by a node appears as a clearly visible left-to-right "wave" that has natural phases that correspond to processing layers of the shortest paths tree. After this animation, most student do not need teacher's explanation that the computation decomposes into N phases of complexity $O(M)$, where N (M, resp.) is the number of nodes (edges, resp.) of the graph.

Similar re-drawing of an input graph in order to visualize invariants is successfully used for some other graph algorithms as well.

Another type of algorithms are those using a Boolean circuit as a computational model. Examples in Algovision are binary addition, Fast Fourier Transform and bitonic sorting network. The last one is a nice example, where we are using three "orthogonal" animations for a single algorithm. First, a recursive construction of the circuit is shown, starting with a single black box with hidden internal structure, which is disclosed in steps until the complete structure of the network is shown. In the same time, at each stage of the construction process, the computation can be animated by showing how the input sequence is transformed step by step at any already disclosed stage into the final sorted sequence. Third, in order to prove properties of the bitonic sorter, we need a mathematical lemma about separation of bitonic sequences that is proved using the third animation.

Proofs of mathematical statements are frequently constructive proofs based on a more or less complex algorithm that is used to construct the entity the existence of which is being demonstrated. A simple way of explaining the proof is by animating the proof algorithm. It is clear that in this case it is not too important what happens, but why it happens, and therefore the ideas explained above are very useful if not necessary.

4. Additional features of educational applets

Visualization of invariants is a feature that is necessary but not sufficient to create an animation useful in algorithm teaching. The termination and correctness proofs are only a part of the stuff that should be explained. E.g., it is quite frequent that certain implementation details like proper or convenient data structures are important.

The Fortune's algorithm for a planar Voronoi diagram uses a "beachline", a sequence of parabolic arcs that presents a boundary of the region where enough information is available to draw segments of the diagram. A search of the beachline arcs can take linear time (wrt the number of sites), but it can be performed in logarithmic time is a binary search tree is built above the set of arcs. Our applet makes it possible to visualize the tree that evolves as the computation (a line sweep) is going on. When the computation is interrupted, a search of a proper arc of the beachline can be animated in the search tree.

In certain cases, small virtual calculators are very helpful. When explaining the discrete Fourier transform, a student can use a spectral calculator that, given an input function described by a mathematical formula or drawn at the display, computes the spectrum of the function. A calculator is very helpful when used in the reverse sense, when a student can set components of the spectrum manually using visual sliding controls, and his or her task is to match the given function. A good understanding of the meaning of the spectrum components can be developed using the calculator.

5. Conclusions

Algorithm visualization is a very useful tool for teaching of algorithms, but only if it does not suggest a simple animation, but it involves additional features that make it possible to visualize anything that is necessary to prove the correctness and termination of the algorithm, implementation issues, applications etc. As a result, visualization systems are very complex software systems that have a number of different components that can be used together.

We strongly believe that there is no uniform way to build such systems, but any algorithm needs a method that is tailored to capture its specific properties. Only if built in this way, it can make teaching of an algorithm easy and/or represent a useful tool for distant learning or e-learning.

6. References:

[Ba81] Baecker, R. (assistance Sherman, D.) Sorting Out Sorting, 30' color film, Morgan Kaufmann Publ., 1981.

[KS96] Kehoe, C. M., Stasko, J. T., Using Animations to Learn about Algorithms: An Ethnographic Case Study, Georgia Institute of Technology Technical Report GIT-GVU-96-20

[KS02] Kerren, A. and Stasko, J., "Algorithm Animation - Introduction", *Software Visualization State of the Art Survey,* Springer Lecture Notes in Computer Science LNCS 2269, Editor: Stephan Diehl, 2002, Chapter 1, pp. 1-15.

[SDBP] Stasko, J., Domingue, J., Brown, M.H., Price, B.A., Software Visualization, MIT Press, 1998.